L69: Entry 1 of 14 File: USPT Jan 27, 2004

DOCUMENT-IDENTIFIER: US 6684261 B1
TITLE: Object-oriented operating system

Detailed Description Text (25):
Threads do have some elements associated with them, however. The containing task and address space, as well as the execution state, have already been discussed. Each thread has a scheduling policy, which determines when and how often the thread will be given a processor on which to run. The scheduling services are discussed in more detail in a later section. Closely tied to the scheduling policy of a thread is the optional processor set designation, which can be used in systems with multiple processors to more closely control the assignment of threads to processors for potentially greater application performance. As indicated before, an address space (task) can contain zero or more threads, which execute concurrently. The kernel makes no assumptions about the relationship of the threads in an address space or, indeed, in the entire system. Rather, it schedules and executes the threads according to the scheduling parameters associated with them and the available processor resources in the system. In particular, there is no arrangement (e.g., hierarchical) of threads in an address space and no assumptions about how they are to interact with each other. In order to control the order of execution and the coordination of threads to some useful end, Mach provides several synchronization mechanisms. The simplest (and coarsest) mechanism is thread-level suspend and resume operations. Each thread has a suspend count, which is incremented and decremented by these operations. A thread whose suspend count is positive remains blocked until the count goes to zero.

Detailed Description Text (28):
The basic organizational entity in Mach for which resources are managed is known as a task. Tasks have many objects and attributes associated with them. A task fundamentally comprises three things. A task contains multiple threads, which are the executable entities in the system. A task also has an address space, which represents virtual memory in which its threads can execute. And a task has a port name space, which represents the valid IPC ports through which threads can communicate with other threads in the system. Each of these fundamental objects in a task is discussed in greater detail in the following sections. Note that a task is not, of itself, an executable entity in Mach. However, tasks can contain threads, which are the execution entities. A task has a number of other entities associated with it besides the fundamental ones noted above. Several of these entities have to do with scheduling decisions the kernel needs to make for the threads contained by the task. The scheduling parameters, processor set designation, and host information all contribute to the scheduling of the task's threads. A task also has a number of distinguished interprocess communication ports that serve certain pre-defined functions. Ports and other aspects of interprocess communication are discussed at length in a later section. For now, it is sufficient to know that port resources are accumulated over time in a task. Most of these are managed explicitly by the programmer. The distinguished ports mentioned above generally have to do with establishing connections to several important functions in the system. Mach supplies three "special" ports with each task. The first is the task self port, which can be used to ask the kernel to perform certain operations on the task. The second special port is the bootstrap port, which can be used for anything (it's OS environment-specific) but generally serves to locate other

services. The third special port that each task has is the host name port, which allows the task to obtain certain information about the machine on which it is running. Additionally, Mach supplies several "registered" ports with each task that allow the threads contained in the task to communicate with certain higher-level servers in the system (e.g., the Network Name Server, the "Service" Server, and the Environment Server).

Detailed Description Text (37):
Threads use ports to communicate with each other. A port is basically a message queue inside the kernel that threads can add messages to or remove message from, if they have the proper permissions to do so. These "permissions" are called port rights. Other attributes associated with a port, besides port rights, include a limit on the number of messages that can be enqueued on the port, a limit on the maximum size of a message that can be sent to a port, and a count of how many rights to the port are in existence. Ports exist solely in the kernel and can only be manipulated via port rights.

Detailed Description Text (39):
A thread can add a message to a port's message queue if it has a send right to that port. Likewise, it can remove a message from a port's message queue if it has a receive right to that port. Port rights are considered to be resources of a task, not an individual thread. There can be many send rights to a port (held by many different tasks); however, there can only be one receive right to a port. In fact, a port is created by allocating a receive right and a port is destroyed only when the receive right is deallocated (either explicitly or implicitly when the task dies). In addition, the attributes of a port are manipulated through the receive right. Multiple threads (on the same or different tasks) can send to a port at the same time, and multiple threads (on the same task) can receive from a port at the same time. Port rights act as a permission or capability to send messages to or receive messages from a port, and thus they implement a low-level form of security for the system. The "owner" of a port is the task that holds the receive right. The only way for another task to get a send right for a port is if it is explicitly given the right--either by the owner or by any task that holds a valid send right for the port. This is primarily done by including the right in a message and sending the message to another task. Giving a task a send right grants it permission to send as many messages to the port as it wants. There is another kind of port right called a send-once right that only allows the holder to send one message to the port, at which time the send-once right become invalid and can't be used again. Note that ownership of a port can be transferred by sending the port's receive right in a message to another task.

Detailed Description Text (46):
The port right disposition array contains the desired processing of the right, i.e., whether it should be copied, made, or moved to the target task. The out-of-line memory disposition array specifies for each memory range whether or not it should be de-allocated when the message is queued, and whether the memory should be copied into the receiving task's address space or mapped into the receiving address space via a virtual memory copy-on-right mechanism. The out-of-line descriptors specify the size, address, and alignment of the out-of-line memory region. When a task receives a message, the header, in-line data, and descriptor arrays are copied into the addresses designated in the parameters to the receive call. If the message contains out-of-line data, then virtual memory in the receiving task's address space is automatically allocated by the kernel to hold the out-of-line data. It is the responsibility of the receiving task to deallocate these memory regions when they are done with the data.

Detailed Description Text (48):
Mach IPC is basically asynchronous in nature. A thread sends a message to a port, and once the message is queued on the port the sending thread continues execution. A receive on a port will block if there are no messages queued on the port. For

efficiency there is a combined send/receive call that will send a message and immediately block waiting for a message on a specified reply port (providing a synchronous model). A time-out can be set on all message operations which will abort the operation if the message is unable to be sent (or if no message is available to be received) within the specified time period. A send call will block if it uses a send-right whose corresponding port has reached its maximum number of messages. If a send uses a send-once right, the message is guaranteed to be queued even if the port is full. Message delivery is reliable, and messages are guaranteed to be received in the order they are sent. Note that there is special-case code in Mach which optimizes for the synchronous model over the asynchronous model; the fastest IPC round-trip time is achieved by a server doing a receive followed by repeated send/receive's in a loop and the client doing corresponding send/receive's in a loop on its side.

Detailed Description Text (60):
Mach also exports the notions of processors and processor sets, which allow tasks to more carefully specify when and on what processors its threads should execute. Processors and processor sets can be enumerated and controlled with the host privilege port. A processor represents a particular processor in the system, and a processor set represents a collection of processors. Services exist to create new processor sets and to add processors to a set or remove them as desired. Services also exist to assign entire tasks or particular threads to a set. Through these services a programmer can control (on a coarse grain) when the threads and tasks that constitute an application actually get to execute. This allows a programmer to specify when certain threads should be executed in parallel in a processor set. The default assignment for tasks and threads that do not explicitly use these capabilities is to the system default processor set, which generally contains any processors in the system that aren't being used in other sets.

Detailed Description Text (156):
TPortReceiveRightHandle represents a port receive right. It supports all the typical operations that can be performed on a receive right, such as requesting no-more-senders notification, setting and getting the port's maximum message size and queue length, getting and setting its make-send count, etc. If a TPortReceiveRightHandle is instantiated (other than with the null or copy constructors) it causes a port and receive right to be created. The copy constructor creates another object (an alias) that references the same receive right. These objects are internally reference counted, such that when the last object referencing a particular receive right is destroyed, it destroys the receive right (and the port) it represents, causing all extant rights to that port to become dead names. This class is a concrete class that represents a port receive right. By definition, the actual kernel port entity is created when a receive right is created, and destroyed when a receive right is destroyed. Since this class is a handle, creation and destruction of the receive right is not necessarily tied to creation and deletion of a TPortReceiveRightHandle. For example, the default constructor does not actually create a receive right, but just an empty object. This class includes Constructors that create a TPortReceiveRightHandle object without creating a port or affecting the kernel reference counts, Constructors that create new Receive Rights and Ports, methods to make an uninitialized object valid, creating a receive right (and therefore a port) in the process, Streaming Support, Receive Right/Port manipulation methods, Getters and setters, and Methods for requesting notifications.

Detailed Description Text (161):
MWaitable and TWaitGroup are classes that provide for message dispatching and the ability to wait for more than one type of message source at the same time. TWaitGroup is a class that provides the ability to set up a collection of objects derived from MWaitable such that a thread can use the Wait method to receive a message from any of the MWaitable objects. It also provides for automatic dispatching of the received message. Multi-Wait Operations are called repeatedly by

a task to receive messages. They are multi thread safe so there can be more than one thread servicing messages. This class includes methods for manipulating the members of the TWaitGroup. For example, GetListOfWaitables returns a list of MWaitables in this group. MWaitable is an abstract base class that associates a port with an internal handler method (HandleIPCMessage). It also provides a common base class for collecting together via the TWaitGroup class Receive Rights and other classes based on Receive Rights

Detailed Description Text (162):
TWaitablePortReceiveRightHandle is a convenience class that derives from both TPortReceiveRightHandle and MWaitable. It is an abstract base class whose subclasses can be added to a TWaitGroup to provide for multi-wait/dispatching of Mach message IPC with other MWaitable subclasses.

Detailed Description Text (181):
TThreadSchedule is a concrete base class that embodies the scheduling behavior of a thread. It defines the thread's actual, default, and maximum priorities. The lower the priority value, the greater the urgency. Each processor set has a collection of enabled TThreadSchedules and a default one. A thread may be assigned any TThreadSchedule that is enabled on the processor set on which the thread is running. The priority may be set up to the maximum value defined by TThreadSchedule, but use of this feature is strongly discouraged. Specific scheduling classes (TIdleSchedule, TServerSchdule etc.) are made available using this class as the base. However (since there are no pure virtual functions in this class) derived classes are free to create objects of this class if necessary (but it may not be required to do so). TThreadSchedule objects (using polymorphism) are used to specify scheduling policy for threads. The subclasses presented below should be used to determine the appropriate priority and proper range.

Detailed Description Text (183):
TServerSchedule is a concrete subclass of TThreadSchedule for server threads. Server threads must be very responsive. They are expected to execute for a short time and then block. For services that take an appreciable amount of time, helper tasks with a different kind of TThreadSchedule (TSupportSchedule) should be used.

Detailed Description Text (206):
TPrivilegedProcessorSetHandle is a concrete class providing the protocol for a processor set control port. Objects of this class can: enable and disable scheduling policies, set the maximum priority for the processor set, return statistics and information, enumerate the tasks and threads, and assign threads and tasks to the processor set. Client access to objects of this class should be highly restricted to protect the individual processors and the processor set.

DOCUMENT-IDENTIFIER: US 6075791 A
TITLE: System for guaranteeing data transfer rates and delays in packet networks

Abstract Text (1):
A system is disclosed which services a plurality of queues associated with respective data connections such that the system guarantees data transfer rates and data transfer delays to the data connections. This is achieved by associating each connection having at least one data packet waiting in its associated queue (such a connection called a backlogged connection) with a timestamp generated as a function of system parameters including (a) the number of queues that are backlogged, (b) the data transfer rate guaranteed to each connection, (c) the sum of data transfer rates guaranteed to all backlogged connections, (d) the previous timestamp of the connection, and (e) the weighted sum of the timestamps of all backlogged connections, each timestamp weighted by the data transfer rate guaranteed to the corresponding connection. The backlogged connection associated with the timestamp having the smallest value among all of the backlogged connections is then identified and a data packet is transmitted from the queue corresponding to that connection. A new timestamp is then generated for that connection if it is still backlogged. Once the transmission of the data packet is completed, the foregoing determination of the connection with the minimum timestamp is then repeated to identify the next queue to be serviced.

Brief Summary Text (2):
The present invention relates to a system for scheduling packets in packet networks and more particularly relates to guaranteeing data transfer rates to data sources and data transfer delays from data sources to destinations.

Brief Summary Text (4):
The provision of negotiated Quality-of-Service (QoS) guarantees such as data transfer rate and data transfer delay to traffic generated by applications of widely different characteristics is a primary objective in packet networks. In such networks, when network resources such as communication links are shared among a plurality of network connections, sophisticated packet scheduling disciplines are necessary to satisfy the QoS requirements of delay-sensitive network connections. A server in a communication system providing such QoS guarantees typically employs multiple queues, in which each queue is associated with a corresponding network connection, and uses a scheduling algorithm to control the order in which the individual queues are served.

Brief Summary Text (5):
One such sophisticated scheduling algorithm is the Generalized Processor Sharing (GPS) scheduling policy. GPS is an idealized scheme which guarantees a negotiated minimum data transfer rate to each network connection, regardless of the behavior of other connections. GPS also guarantees a negotiated maximum end-to-end data transfer delay to each connection, and ensures that all connections are served in a fair manner. During any time interval, a GPS server serves all backlogged queues, i.e., queues which have packets waiting to be transmitted, simultaneously, each with an instantaneous service rate that is a function of the negotiated data transfer rate and delay associated with the respective connection. It can be appreciated that, because all connections have to be serviced simultaneously, the

GPS algorithm cannot be implemented in a real packet network, and, therefore. is considered to be an ideal scheduling discipline.

Brief Summary Text (6):
A class of scheduling disciplines, called GPS-related packet-scheduling disciplines, which approximate the GPS scheduling policy to a certain degree, may be implemented in a real packet network. These algorithms are all based on maintaining a global function, referred to as virtual time or as system potential, which is a measure of the amount of service that has been provided by a server. A server uses this global function to compute a finishing virtual time, also referred to as timestamp, for each packet in the associated system; in which the timestamp is indicative of when the service of the corresponding packet should be completed. The server serves the packets based on the values of their respective timestamps, starting with the smallest value. The GPS-related packet-scheduling disciplines differ from one another in the specific function used as virtual time. Similarly to GPS, all the GPS-related packet-scheduling disciplines provide a negotiated minimum data transfer rate to each network connection. The specific virtual-time function that is used by each such discipline determines the implementation complexity of the algorithm, the value of the maximum data transfer delay that the algorithm can provide to a connection, and whether or not the algorithm serves all connections in a fair manner. An algorithm that is of minimum complexity, provides a small value of the maximum data transfer delay to each connection, and serves all connections fairly, is highly desirable.

Brief Summary Text (7):
One example of a GPS-related packet-scheduling discipline is the so called Packet-by-packet GPS (P-GPS) algorithm. P-GPS guarantees maximum data transfer delays very close to those of GPS, and serves the connections almost as fairly as GPS. P-GPS uses a system-potential function that is directly derived from a background simulation of GPS. However, P-GPS is not practical in high-speed packet networks, since the task of simulating GPS is very complex. Another example of a GPS-related packet-scheduling discipline is a so called Self-Clocked Fair Queueing (SCFQ) algorithm. SCFQ has minimum implementation complexity, which makes it practical for implementation in high-speed packet networks. It also serves the connections almost as fairly as GPS. However, SCFQ guarantees maximum data transfer delays that are much larger than those that are guaranteed by GPS and P-GPS. Another example of a GPS-related packet-scheduling discipline is a so called Virtual Clock algorithm. The Virtual Clock algorithm has a minimum implementation complexity; and it also guarantees maximum data transfer delays very close to those of GPS and P-GPS. However, it does not always serve all connections in a fair manner. Still another example of a GPS-related packet-scheduling discipline is a Starting-Potential Fair Queueing (SPFQ) algorithm. The SPFQ algorithm guarantees maximum data transfer delays very close to those of GPS and P-GPS, and serves the connections almost as fairly as GPS and P-GPS. However, the implementation complexity of the SPFQ algorithm is considerably higher than either of the SCFQ and Virtual Clock algorithms.

Brief Summary Text (12):
In accordance with a first aspect of the invention, which pertains to the case in which all packets arriving at the scheduling system have equal length, the scheduling of the packets is performed by generating a timestamp for the packet at the head of each of the queues associated with respective connections that have packets awaiting in the scheduling system to be scheduled for transmission (such connections are referred herein to as backlogged connections), and by selecting for transmission the packet whose timestamp has the minimum value among the timestamps for all packets that are at the head of. all the queues associated with backlogged connections. The timestamp for each packet associated with a connection i is generated at the time when the packet reaches the head of the queue corresponding to connection i, and is done as a function of (a) the system potential at that time, (b) the value of the timestamp of the previous packet of connection i, and

(c) the data transfer rate allocated to connection i. The system potential is determined as a function of (a) the number of all connections that are backlogged in the scheduling system at that time, (b) the sum of the rates of all the connections that are backlogged in the scheduling system at that time, and (c) the weighted average of the timestamps of the packets that are waiting for transmission at the head of the queues of all backlogged connections in the scheduling system at that time, the weight of each timestamp being a function of the data transfer rate allocated to the corresponding connection.

Brief Summary Text (13):
In accordance with another aspect of the invention, which pertains to the case in which the packets arriving at the scheduling system have different lengths, the scheduling of the packets is performed by generating a timestamp for the packet at the head of each of the queues associated with backlogged connections, and by selecting for transmission the packet whose timestamp has the minimum value among the timestamps for all packets that are at the head of all the queues associated with backlogged connections. The timestamp for each packet of a connection i is generated at the time when the packet reaches the head of the queue corresponding to connection i, and is done as a function of (a) the system potential at that time, (b) the value of the timestamp of the previous packet of connection i, (c) the length of the packet, and (d) the data transfer rate allocated to connection i. The system potential in this case is determined as a function of (a) the timestamps of the packets that are waiting for transmission at the head of the queues of all the connections that are backlogged in the scheduling system at that time, (b) the sum of the rates of all the connections that are backlogged in the scheduling system at that time, (c) the sum of length of all the packets that are waiting for transmission at the head of the queues of all the connections that are backlogged in the scheduling system at that time, and (d) the weighted average of the timestamps of the packets that are waiting for transmission at the head of the queues of all backlogged connections in the scheduling system at that time, the weight of each timestamp being a function of the data transfer rate allocated to the corresponding connection.

Drawing Description Text (3):
FIG. 1 illustrates a packet network in which a number of switches, data sources and destinations are connected.

Drawing Description Text (4):
FIG. 2 illustrates a communication switch in the packet network of FIG. 1.

Drawing Description Text (6):
FIG. 4 is a block diagram of the server that is part of the communication link interface of FIG. 3.

Drawing Description Text (10):
FIG. 8 is a block diagram of the server that is part of the communication link interface of FIG. 7.

Detailed Description Text (2):
FIG. 1 shows a packet network in which a plurality of switches 1-1 through 1-p are connected to each other by communication links. A number of data sources 2-1 through 2-q are connected to the communication switches. A network connection is established from each of the data sources to a corresponding destination 3-1 through 3-q, and data packets are transmitted from each data source to the corresponding destination.

Detailed Description Text (3):
FIG. 2 shows a communication switch in such packet network, e.g., switch 1-1. The communication switch includes, inter alia, a plurality of communication link interfaces 500-1 through 500-s. Each of said communication link interfaces connects

a plurality of input links to an output link; the communication link interface receives the data packets associated with corresponding network connections from the input links and transmits them to the output link. As shown in FIG. 2, the communication switch may contain just one or a plurality of such communication link interfaces 500. For example, such a communication link interface 500 may be in front of the switch fabric 550, in which case the input links of the communication link interface may be a plurality of input links of the communication switch, and the output link of the communication link interface connects to the switch fabric 550, or such a communication link interface 500 may be at the output of the switch fabric 550, in which case the input links of the communication link interface may be a plurality of output links of the switch fabric 550, and the output link of the communication link interface may be one of the output links of the communication switch.

Detailed Description Text (5):
The communication link interface 500 includes a data packet receiver 10 which receives the data packets arriving from input links 5-1 through 5-m. Receiver 10 uses the contents of a connection identifier field contained in the header of each packet (not shown) to identify its respective connection i. In this embodiment, all packets that the receiver 10 receives have the same length. For example, this is the case when the switch that contains the communication link interface is connected to an Asynchronous Transfer Mode (ATM) network. For each packet, the receiver 10 also determines, at the time of receiving a packet, whether or not the packet can be queued in a connection queue 20-i corresponding to the identified connection i, as will be described below. If the packet can be queued, then the receiver 10 stores the packet in the appropriate connection queue 20-i. Server 100 (described below), in response to that action, increments the queue length register 60-i associated with the identified queue; otherwise, if receiver 10 determines that the packet cannot be queued, the packet is discarded.

Detailed Description Text (6):
For each connection i of a plurality of switched connections 1 through n, the communication link interface 500 includes (a) a connection queue 20-i, which may be, for example, a so-called First-In-First-Out (FIFO) queue used to store the received data packets of connection i, (b) a connection identifier register 30-i, used to store the local identifier identifying connection i, (c) a rate register 40-i, used to store the value of data transfer rate reserved to connection i, (d) a timestamp register 50-i, used to store the timestamp of connection i, and (e) a queue length register 60-i, used to store the number of data packets of connection i that are currently in the communication link interface 500 (including the transmitter 200). When a packet associated with connection i is being transmitted by transmitter 200, the timestamp of connection i is the timestamp of that packet. Also, when connection queue 20-i is not empty and the transmitter 200 is not transmitting a packet associated with connection i, the timestamp of connection i is the timestamp of the packet at the head of connection queue 20-i. Further, when connection queue 20-i is empty and the transmitter 200 is not transmitting a packet associated with connection i, the timestamp of connection i is the timestamp of the last transmitted packet of connection i.

Detailed Description Text (7):
Server 100 generates a new timestamp for connection i every time a new packet reaches the head of the associated connection queue 20-i, and stores the value of the newly-generated timestamp in timestamp register 50-i. Every time the transmitter 200 becomes available for transmission of a next packet, Server 100 selects a data packet among all the packets waiting at the head of the queues associated with all the connections that are backlogged at the time of such availability, and sends the selected packet to the transmitter 200. The assignment of a new timestamp to a connection i is based on the following rules, which depend on whether or not the connection i was backlogged before the new packet arrived at the head of connection queue 20-i.

Detailed Description Text (8):
If connection i was not backlogged before the packet arrived at the head of connection queue 20-i, meaning that connection i becomes backlogged because of the arrival of the new packet, then the new timestamp of connection i is generated according to the following expression: ##EQU1## where F.sub.i.sup.k is the timestamp assigned to the k-th data packet of connection i (F.sub.i.sup.0 =0), a.sub.i.sup.k is the time of arrival of the k-th data packet of connection i at the head of connection queue 20-i, P(a.sub.i.sup.k) is the value of the system potential at time a.sub.i.sup.k and is maintained by Server 100, and r.sub.i is the data transfer rate allocated to connection i, normalized to the rate of Server 100.

Detailed Description Text (9):
If connection i was backlogged before the packet arrived at the head of connection queue 20-i, meaning that the new packet has arrived at the head of connection queue 20-i as a result of transmitter 200 having just transmitted another packet of connection i, then the new timestamp of connection i is assigned according to the following equation: ##EQU2## where F.sub.i.sup.k-1 is the timestamp assigned to the (k-1)-th data packet of connection i.

Detailed Description Text (10):
At the end of the transmission of the j-th packet in the system, the system-potential function P appearing in equation (1) and maintained by server 100 is computed according to the following equation: ##EQU3## where:

Detailed Description Text (11):
B(j) is the set of all the backlogged connections at that time, which are all the connections i whose corresponding queue length register 60-i contains a number greater than 0 at the end of transmission of the j-th packet in the system. ##EQU4## is the weighted sum of the timestamp values F.sub.i that are stored in the timestamp registers 50-i of all the backlogged connections i.epsilon.B(j) at the end of the transmission of the j-th packet in the system. In the weighted sum, the weight of each timestamp value F.sub.i is the reserved data transfer rate r.sub.i of the corresponding connection i; the value of r.sub.i is stored in rate register 40-i.

Detailed Description Text (14):
Server 100 determines the system-potential function each time a transmission is completed. Whenever server 100 becomes idle as a result of all connection queues 20-1 through 20-n being empty, then server 100 resets to 0 the system potential P and the content of each timestamp register 50-i.

Detailed Description Text (15):
Whenever transmitter 200 is available for the transmission of a new/next data packet, then server 100 selects the next connection i to be serviced. Connection i is selected for transmission if the content of queue length register 60-i is greater than 0, and the content of timestamp register 50-i has a value that is the minimum among the values contained in the timestamp registers 50-h associated with all backlogged connections h.

Detailed Description Text (16):
When a connection i is selected for transmission by Server 100, then the data packet at the head of the corresponding connection queue 20-i is unloaded from that queue and supplied to the transmitter 200.

Detailed Description Text (17):
When the transmitter 200 completes the transmission over the output link 300 of a packet belonging to connection i, Server 100 decrements the content of the queue length register 60-i. If the decremented value stored in queue length register 60-i

is greater than 0, then Server 100 generates the new timestamp of connection i. Then Server 100 updates the system-potential function P according to equation (3) using (a) the value of the newly-generated timestamp of connection i, (b) the previous value of the timestamp of connection i stored in timestamp register 50-i, and (c) the value in rate register 40-i. Then, Server 100 removes the previous value in timestamp register 50-i, and stores the value of the newly-generated timestamp of connection i in the timestamp register 50-i. If the decremented value stored in queue length register 60-i is 0, then Server 100 updates the system-potential function P according to equation (3) using the content of timestamp register 50-i and the content of rate register 40-i.

Detailed Description Text (18):
FIG. 4 shows an apparatus or server 100 according to the first illustrative embodiment of the present invention.

Detailed Description Text (19):
The first embodiment of server 100 includes (a) a register 110, for storing the sum of the reserved data transfer rates of all backlogged connections (the content of register 110 is called r.sub.B(j) in equation (3) above); (b) a register 115, for storing the number of backlogged connections (the content of register 115 is called n.sub.B(j) in equation (3) above); (c) a register 120, for storing the current value of the system-potential function (the content of register 120 is called P in equation (1), equation (2), and equation (3) above); and (d) a register 123, for storing the weighted sum of the timestamps of all backlogged connections (the content of register 123 is called F.sub.B(j) in equation (2)). Server 100 also includes (a) a controller 130, which (i) updates the content of registers 110, 115, and 123, (ii) supplies the content of registers 110, 115, and 123 to controller 135, and (iii) supplies to controller 140 the previous/old value of the timestamp stored in the timestamp register 50-i of the connection i for which a new timestamp must be computed; (b) a controller 135, which determines the value of the system potential according to equation (3) each time a transmission of a packet associated with connection i in the system is completed by transmitter 200, using the content of registers 110, 115, 120, 123, and 50-i, and the new value of

Detailed Description Text (20):
the timestamp of connection i; and (c) a controller 140, which determines the timestamp to be assigned to connection i when a new packet arrives at the head of its connection queue 20-i, according to (i) equation (1) if connection i was not backlogged before the new packet arrived at the head of connection queue 20-i, and (ii) equation (2) if connection i was already backlogged before the new packet arrived at the head of the connection queue 20-i.

Detailed Description Text (21):
Server 100 further includes a sorter 160 and a selector 170. At any time the transmitter 200 becomes available for a new transmission of a data packet, the sorter 160 supplies the identifier of the backlogged connection i whose timestamp register 50-i contains the minimum value among all backlogged connections. The selector 170 removes from connection queue 20-i the packet that is at the head of connection queue 20-i corresponding to connection i, whose identifier has been supplied by the sorter 160, and supplies the packet to the transmitter 200.

Detailed Description Text (22):
FIGS. 5A through 5C (which are arranged relative to one another as shown in FIG. 6) illustrate in flow chart form showing an illustrative embodiment of the operation of server 100 that implements a first method of scheduling the transmission of data packets of fixed size according to the present invention.

Detailed Description Text (23):
Referring to FIGS. 5A-5C, in step S510, if new data packets have arrived at the receiver 10, then server 100 proceeds to step S550. Otherwise, Server 100 proceeds

to step S520.

Detailed Description Text (24):
In step S520, if no backlogged connections are available (which is indicated by the content of register 115 being equal to 0), server 100 then proceeds to step S530. Otherwise, Server 100 proceeds to step S670.

Detailed Description Text (25):
In step S530, Server 100 resets to 0 the content of register 120 (which stores the value of the system potential) and all of the registers 50-1 through 50-n. Server 100 then proceeds to step S510.

Detailed Description Text (26):
In step S550, Server 100 selects one of the data packets that have just arrived at the receiver 10. Server 100 then proceeds to step S560.

Detailed Description Text (27):
In step S560, Server 100 identifies the connection i corresponding to the packet selected in step S550. Server 100 identifies connection i through a connection identifier contained in the header of the received packet (not shown). The identification of connection i allows to identify connection queue 20-i corresponding to connection i, where the packet should be stored. Server 100 then proceeds to step S570.

Detailed Description Text (28):
In step S570, Server 100 stores the packet in connection queue 20-i and then proceeds to step S580.

Detailed Description Text (29):
In step S580, if the content of queue length register 60-i is 0 (connection i is not backlogged), Server 100 proceeds to step S590. Otherwise, Server 100 proceeds to step S970.

Detailed Description Text (30):
In step S590, server 100 (more specifically, the controller 130) increments the content of register 115 and then proceeds to step S600.

Detailed Description Text (31):
In step S600, server 100 increments the content of queue length register 60-i. Server 100 then proceeds to step S610.

Detailed Description Text (32):
In step S610, server 100 (more specifically, the controller 140) computes the new timestamp of connection i identified in step S560 according to equation (1). Server 100 then proceeds to step S615.

Detailed Description Text (33):
In step S615, server 100 stores the timestamp of connection i in timestamp register 50-i and then proceeds to step S620.

Detailed Description Text (34):
In step S620, server 100 (more specifically, the controller 130) adds the content of rate register 40-i to the content of register 110. Server 100 then proceeds to step S640.

Detailed Description Text (35):
In step S640, server 100 (more specifically, the controller 130) adds the product of the contents of timestamp register 50-i and rate register 40-i to the content of register 123. Server 100 then proceeds to step S660.

Detailed Description Text (36):
In step S660, if additional new data packets have been received at the receiver 10, then server 100 proceeds to step S550. Otherwise, server 100 proceeds to step S670.

Detailed Description Text (37):
In step S670, the sorter 160 identifies the minimum among the contents of all registers 50-j corresponding to backlogged connections j. Server 100 then proceeds to step S680.

Detailed Description Text (38):
In step S680, the selector 170 identifies the connection i corresponding to the minimum timestamp found in step S670. Server 100 then proceeds to step S690.

Detailed Description Text (39):
In step S690, if transmitter 200 is already transmitting a packet over output link 300 and is therefore not available to transmit an other packet, then server 100 proceeds to step S800. Otherwise, server 100 proceeds to step S700.

Detailed Description Text (40):
In step S700, if the content of register 120, which stores the value of the system potential, is equal to 0 server 100 proceeds to step S780. Otherwise, server 100 proceeds to step S710.

Detailed Description Text (41):
In step S710, server 100 decrements the queue length register 60-h for the connection h corresponding to the last packet being transmitted by transmitter 200, and then proceeds to step S720.

Detailed Description Text (42):
In step S720, if the content of queue length register 60-h of connection h corresponding to the last packet being transmitted by transmitter 200 is 0, then server 100 proceeds to step S722. Otherwise, server 100 proceeds to step S730.

Detailed Description Text (43):
In step S722, server 100 (more specifically, the controller 130) decrements the content of register 115. Server 100 then proceeds to step S724.

Detailed Description Text (44):
In step S724, server 100 (more specifically, the controller 130) subtracts from the content of register 110 the content of rate register 40-h of connection h corresponding to the last packet being transmitted by transmitter 200. Server 100 then proceeds to step S728.

Detailed Description Text (45):
In step S728, server 100 (more specifically, the controller 130) subtracts the product of the contents of timestamp register 50-h and rate register 40-h of connection h corresponding to the last packet being transmitted by transmitter 200 from the content of register 123. Server 100 then proceeds to step S780.

Detailed Description Text (46):
In step S730, server 100 (more specifically, the controller 130) subtracts the product of the contents of timestamp register 50-h and rate register 40-h of connection h corresponding to the last packet being transmitted by transmitter 200 from the content of register 123. Server 100 then proceeds to step S735.

Detailed Description Text (47):
In step S735, server 100 (more specifically, the controller 140) computes the new timestamp of connection h corresponding to the last packet being transmitted by transmitter 200 according to equation (2). Server 100 then proceeds to step S740.

Detailed Description Text (48):
In step S740, server 100 stores the value of the newly-generated timestamp of connection h corresponding to the last packet being transmitted by transmitter 200 in timestamp register 50-h, and then proceeds to step S750.

Detailed Description Text (49):
In step S750, server 100 (more specifically, the controller 130) adds the product of the contents of timestamp register 50-h and rate register 40-h of connection h corresponding to the last packet being transmitted by transmitter 200 to the content of register 123. Server 100 then proceeds to step S780.

Detailed Description Text (50):
In step S780, the packet at the head of connection queue 20-i corresponding to connection i identified in step S680 by server 100 (more specifically, by the selector 170) is sent to the transmitter 200. Server 100 then proceeds to step S790.

Detailed Description Text (51):
In step S790, server 100 (more specifically, the controller 135) computes the new value of the system potential according to equation (3), using the content of registers 110, 115, 120, and 123, and stores it in register 120. Server 100 then proceeds to step S510.

Detailed Description Text (52):
In step S800, if new data packets have arrived at the receiver 10, then server 100 proceeds to step S820. Otherwise, server 100 proceeds to step S690.

Detailed Description Text (53):
In step S820, Server 100 selects one of the data packets that have just arrived at the receiver 10. Server 100 then proceeds to step S830.

Detailed Description Text (54):
In step S830, Server 100 identifies the connection w corresponding to the packet selected in step S820. Server 100 identifies the connection w through a connection identifier contained in the header of the packet (not shown). The identification of connection w allows server 100 to identify the connection queue 20-w corresponding to connection w where the packet should be stored. Server 100 then proceeds to step S840.

Detailed Description Text (55):
In step S840, server 100 stores the packet selected in step S820 in connection queue 20-w and then proceeds to step S850.

Detailed Description Text (56):
In step S850, if the content of queue length register 60-w is 0 (connection w is not backlogged), Server 100 proceeds to step S860. Otherwise, server 100 proceeds to step S940.

Detailed Description Text (57):
In step S860, server 100 (more specifically, the controller 130) increments the content of register 115. Server 100 then proceeds to step S870.

Detailed Description Text (58):
In step S870, server 100 increments the content of queue length register 60-w, and then proceeds to step S880.

Detailed Description Text (59):
In step S880, server 100 (more specifically, the controller 140) computes the new timestamp of connection i identified in step S830 according to equation (1). Server

100 then proceeds to step S885.

Detailed Description Text (60):
In step S885, server 100 stores the timestamp of connection w in timestamp register 50-w, and 100 then proceeds to step S890.

Detailed Description Text (61):
In step S890, server 100 (more specifically, the controller 130) adds the content of rate register 40-w to the content of register 110. Server 100 then proceeds to step S910.

Detailed Description Text (62):
In step S910, server 100 (more specifically, the controller 130) adds the product of the contents of timestamp register 50-w and rate register 40-w to the content of register 123. Server 100 then proceeds to step S930.

Detailed Description Text (63):
In step S930, if additional new data packets are available, then server 100 proceeds to step S820. Otherwise, server 100 proceeds to step S670.

Detailed Description Text (64):
In step S940, server 100 increments the content of queue length register 60-w corresponding to connection w identified in step S830, and then proceeds to step S930.

Detailed Description Text (65):
In step S970, server 100 increments the content of queue length register 60-i corresponding to connection i identified in step S560, and then proceeds to step S660.

Detailed Description Text (67):
The communication link interface 1500 includes a receiver 1010 which receives the data packets arriving from input links 1005-1 through 1005-m and for each packet identifies its respective connection i through a connection identifier field contained in the header of the packet (not shown); thereafter, a local connection identifier which identifies the corresponding connection is determined for each packet. The receiver 1010 also determines the length of each received packet using length information contained in the packet header. For each packet, the receiver 1010 also determines whether or not the packet can be queued to connection queue 1020-i corresponding to connection i (described below). If the received packet can be queued, then receiver 1010 stores the packet in the associated connection queue 1020-i, and stores the determined packet length in the associated packet length queue 1026-i (described below). Server 100 (described below) then increments the queue length register 1060-i. If the received packet cannot be queued, then the packet is discarded.

Detailed Description Text (68):
For each connection i of a plurality of switched connections 1 through n, the communication link interface 1500 includes (a) a connection queue 1020-i, which may be a FIFO queue used to store the data packets of connection i, (b) a packet length queue 1026-i, which may be a FIFO queue used to store the lengths of the data packets of connection i, (c) a connection identifier register 1030-i, used to store the local identifier identifying connection i, (d) a rate register 1040-i, used to store the value of data transfer rate reserved to connection i, (e) a timestamp register 1050-i, used to store the timestamp of connection i, and (f) a queue length register 1060-i, used to store the number of data packets of connection i that are currently in the communication link interface 1500 (including transmitter 1200). When a packet associated with connection i is being transmitted by transmitter 1200, the timestamp of connection i is the timestamp of that packet. Also, when connection queue 1020-i is not empty and transmitter 1200 is not

transmitting a packet associated with connection i, the timestamp of connection i is the timestamp of the packet at the head of connection queue 1020-i. Further, when connection queue 1020-i is empty and transmitter 1200 is not transmitting a packet associated with connection i, the timestamp of connection i is the timestamp of the last transmitted packet of connection i.

Detailed Description Text (69):
Server 1100 generates a new timestamp for connection i every time a new packet reaches the head of connection queue 1020-i, and stores the value of the newly-generated timestamp in timestamp register 1050-i. Each time transmitter 1200 becomes available for transmission of a new/next packet, server 1100 selects a data packet among all the packets waiting at the head of the queues associated with all the connections that are backlogged at the time of such availability, and sends the selected packet to transmitter 1200. The assignment of a new timestamp to a connection i is based on the following rules, which depend on whether or not the connection i was backlogged before the new packet arrived at the head of connection queue 1020-i.

Detailed Description Text (70):
If connection i was not backlogged before the packet arrived at the head of connection queue 1020-i, meaning that connection i becomes backlogged because of the arrival of the new packet, the new timestamp that is assigned to connection i is generated according to the following expression: ##EQU5## where F.sub.i.sup.k is the timestamp assigned to the k-th data packet of connection i (F.sub.i.sup.0 =0), a.sub.i.sup.k is the time of arrival of the k-th data packet of connection i at connection queue 1020-i, P(a.sub.i.sup.k) is the value of the system potential at time a.sub.i.sup.k and is maintained by server 1100, l.sub.i.sup.k is the length of the k-th data packet of connection i, and r.sub.i is the data transfer rate allocated by server 1100 to connection i, normalized to the rate of server 1100.

Detailed Description Text (71):
If connection i was backlogged before the packet arrived at the head of connection queue 1020-i, meaning that the new packet has arrived at the head of connection queue 1020-i as a result of transmitter 1200 having transmitted an other packet of connection queue i, the new timestamp to connection i is assigned according to the following expression: ##EQU6## where F.sub.i.sup.k-1 is the timestamp assigned to the (k-1)-th data packet of connection i.

Detailed Description Text (72):
The system-potential function P appearing in equation (4) and maintained by server 1100 is determined at each time tj when the end of the transmission of the j-th packet in the system occurs. The system-potential function P is computed at time tj according to the following equation: ##EQU7## where:

Detailed Description Text (73):
B(t.sub.j) is the set of all the backlogged connections at time tj, which are all the connections i whose corresponding queue length register 1060-i contains a number greater than 0 at time tj. ##EQU8## is the weighted sum of the timestamp values F.sub.i that are stored in the timestamp registers 1050-i of all the backlogged connections i.epsilon.B(j) at time tj. In the weighted sum, the weight of each timestamp value F.sub.i is the reserved data transfer rate r.sub.i of the corresponding connection i; the value of r.sub.i is stored in rate register 1040-i. ##EQU9## is the sum of the values that are stored at the head of the packet length queues 1026-i at time tj computed over all the connections i.epsilon.B(t.sub.j). ##EQU10## is the sum of the values that are stored in the rate registers 1040-i,

Detailed Description Text (76):
Server 1100 determines the system-potential function each time such a transmission is completed. Server 1100 also computes the system-potential function every time the k-th packet arrives at the head of connection queue 1020-i and connection i was

not backlogged before the k-th packet arrived, and uses it to compute the new timestamp of connection i according to equation (4). In this case, server 1100 computes the value of P(a.sub.i.sup.k) which appears in equation (4) according to the following expression:

Detailed Description Text (77):
where a.sub.i.sup.k is the time of arrival of the k-th data packet of connection i at the head of connection queue 1020-i, P(t.sub.h) is the last value of the system potential computed by server 1100 before a.sub.i.sup.k according to equation (6), where P(t.sub.h) was computed at the time t.sub.h corresponding to the end of the last transmission of a packet by transmitter 1200 which occurred before a.sub.i.sup.k.

Detailed Description Text (78):
Each time server 1100 becomes idle as a result of all the connection queues 1020-1 through 1020-n being empty, Server 100 resets to 0 the system potential P and the content of each timestamp register 1050-i.

Detailed Description Text (79):
Each time transmitter 1200 becomes available for the transmission of a new/next packet, server 1100 selects the next connection i to be serviced. Connection i is selected for transmission if the content of queue length register 1060-i is greater than 0 and the content of timestamp register 1050-i has a value that is the minimum among the values contained in the timestamp registers 1050-h associated with all backlogged connections h.

Detailed Description Text (80):
When a connection i is selected for transmission by server 1100, then the data packet at the head of the corresponding connection queue 1020-i is unloaded from that queue and supplied to transmitter 1200.

Detailed Description Text (81):
When transmitter 1200 completes the transmission over the output link 1300 of a packet belonging to connection i, server 1100 decrements the content of queue length register 1060-i. If the decremented value stored in the queue length register 1060-i is greater than 0, then server 1100 removes the old value at the head of packet length queue 1026-i, and the value at the head of packet length queue 1026-i becomes the next value in packet length queue 1026-i. Then, server 1100 generates the new timestamp for connection i. Server 1100 then updates the system-potential function P according to equation (6) using (a) the value of the newly-generated timestamp of connection i, (b) the previous value of the timestamp of connection i stored in timestamp register 1050-i, (c) the old value at the head of packet length queue 1026-i, (d) the new value at the head of packet length queue 1026-i, (e) and the value in rate register 1040-i. Then, server 1100 removes the previous value in timestamp register 1050-i, and stores the value of the newly-generated timestamp of connection i in timestamp register 1050-i. If the decremented value stored in queue length register 1060-i is 0, then server 1100 updates the system-potential function P according to equation (6) using (a) the content of timestamp register 1050-i, (b) the content of rate register 1040-i, and (c) the value at the head of packet length queue 1026-i. Then, server 1100 removes the previous value at the head of packet length queue 1026-i.

Detailed Description Text (82):
FIG. 8 shows an illustrative embodiment of an apparatus or server 1100 according to a second illustrative embodiment of the present invention.

Detailed Description Text (83):
Specifically, server 1100 includes (a) a register 1110, for storing the sum of the reserved service rates of all backlogged connections (the content of register 1110 is called r.sub.B(tj) in equation (6) above); (b) a register 1115, for storing the

number of backlogged connections (the content of register 1115 is called n.sub.B (tj) in equation (6) above); (c) a register 1117, for storing the sum of the packet lengths that are currently at the head of packet length queues 1026-1 through 1026-n (the content of register 1117 is called L.sub.B(tj) in equation (6) above); (d) a register 1120, for storing the current value of the system-potential function (the content of register 1120 is called P in equation (4), equation (6), and equation (7)); (e) a register 1123, for storing the weighted sum of the timestamps of all backlogged connections (the content of register 1123 is called F.sub.B(tj) in equation (6) above); and (f) a counter 1127, for storing the time elapsed since the end of the latest transmission of a packet in the system by transmitter 1200 (the counter 1127 is used to compute the value of (tj-tj-1) which appears in equation (6) above and the value of (a.sub.i.sup.k -tj) which appears in equation (7) above). Server 1100 also includes (a) a controller 1130, which (i) updates the contents of registers 1110, 1115, 1117 and 1123, (ii) supplies the content of registers 1110, 1117 and 1123 to controller 1135, and (iii) supplies to controller 1140 the old value of the timestamp stored in the timestamp register 1050-i of the connection i for which a new timestamp must be computed; (b) a register 1132, used to store the length of the last received packet; (c) a controller 1135 which determines, using the contents of registers 1110, 1117, 1120, 1123 and 1127, and 1050-i, and the new value of the timestamp of connection i, the value of the system potential according to equation (6) each time transmitter 1200 completes the transmission of a packet associated with connection i in the system, and then resets the content of register 1127. Controller 1135 also determines the system potential according to equation (7) each time a packet arrives at the head of its connection queue 1020-i when connection i is not backlogged, and provides the value of the system potential (determined according to equation (7) to controller 1140. Server 100 also includes a controller 1140, which determines the timestamp to be assigned to connection i when a new packet arrives at the head of its connection queue 1020-i, according to (i) equation (4) if connection i was not backlogged before the new packet arrived at the head of connection queue 1020-i, and (ii) equation (5) if connection i was already backlogged before the new packet arrived at the head of the connection queue 1020-i.

Detailed Description Text (84):
Server 110 further includes a sorter 1160 and a selector 1170. Whenever transmitter 1200 becomes available for a new transmission, then sorter 1160 supplies the identifier of the backlogged connection i whose timestamp register 1050-i contains the minimum value among all backlogged connections. Selector 1170 removes from connection queue 1020-i the packet that is at the head of connection queue 1020-i corresponding to connection i, whose identifier has been supplied by sorter 1160, and supplies the packet to transmitter 1200.

Detailed Description Text (85):
FIGS. 9A through 9C (which should be arranged with respect to one another as shown in FIG. 10) show in flow chart form the way in which server 1100 a second illustrative method of scheduling the transmission of data packets of different sizes according to the present invention. Note that FIGS. 9A-9C are somewhat similar to FIGS. 5A-5C. As such, the following discussion of FIGS. 9A-9C is somewhat similar to the discussion of FIGS. 5A-5C, with the exception that the discussion of FIGS. 9A-9C is directed to, among other things, equations (4) through (7), and includes additional operation steps directed to the processing of the length of the received packets.

Detailed Description Text (86):
Referring to FIGS. 9A-9C, in step S1510, if new data packets have arrived at receiver 1010, then server 1100 proceeds to step S1550. Otherwise, server 1100 proceeds to step S1520.

Detailed Description Text (87):
In step S1520, if no backlogged connections are available (which is indicated by

the content of register 1115 being equal to 0), server 1100 proceeds to step S1530. Otherwise, server 1100 proceeds to step S1670.

Detailed Description Text (88):
In step S1530, server 1100 resets to 0 the content of register 1120 (which stores the value of the system potential), of counter 1127, and all of the registers 1050-1 through 1050-n. Server 1100 then proceeds to step S1510.

Detailed Description Text (89):
In step S1550, server 1100 selects one of the data packet just arrived at the receiver 1010, and then proceeds to step S1560.

Detailed Description Text (90):
In step S1560, server 1100 identifies the connection i corresponding to the packet selected in step S1550. Server 1100 identifies connection i using a connection identifier contained in the header of the received packet (not shown). The identification of connection i allows to identify connection queue 1020-i corresponding to connection i, where the packet should be stored. Server 1100 then proceeds to step S1570.

Detailed Description Text (91):
In step S1570, server 1100 stores the packet in connection queue 1020-i, and then proceeds to step S1575.

Detailed Description Text (92):
In step S1575, server 1100 stores the length of the packet in packet length queue 1026-i and in register 1132, and then proceeds to step S1580.

Detailed Description Text (93):
In step S1580, if the content of queue length register 1060-i is 0 (connection i is not backlogged), then server 1100 proceeds to step S1590. Otherwise, server 1100 proceeds to step S1970.

Detailed Description Text (94):
In step S1590, server 1100 (more specifically, the controller 1130) increments the content of register 1115. Server 1100 then proceeds to step S1600.

Detailed Description Text (95):
In step S1600, server 1100 increments the content of queue length register 1060-i, and then proceeds to step S1610.

Detailed Description Text (96):
In step S1610, server 1100 (more specifically the controller 1135) computes the value of P(a.sub.i.sup.k), which appears in equation (4), according to equation (7), and provides the value to controller 1140. Then, server 1100 (more specifically, controller 1140) determines the new timestamp of connection i identified in step S1560 according to equation (4). Server 1100 then proceeds to step S1615.

Detailed Description Text (97):
In step S1615, server 1100 stores the timestamp of connection i in timestamp register 1050-i, and then proceeds to step S1620.

Detailed Description Text (98):
In step S1620, server 1100 (more specifically, the controller 1130) adds the content of rate register 1040-i to the content of register 1110. Server 1100 then proceeds to step S1630.

Detailed Description Text (99):
In step S1630, server 1100 (more specifically, the controller 1130) adds the

content of register 1132 to the content of register 1117. Server 1100 then proceeds to step S1640.

Detailed Description Text (100):
In step S1640, server 1100 (more specifically, the controller 1130) adds the product of the contents of timestamp register 1050-i and rate register 1040-i to the content of register 1123. Server 1100 then proceeds to step S1660.

Detailed Description Text (101):
In step S1660, if additional new data packets have been received at the receiver 1010, then server 1100 proceeds to step S1550. Otherwise, server 1100 proceeds to step S1670.

Detailed Description Text (103):
In step S1680, selector 1170 identifies the connection i corresponding to the minimum timestamp found in step S1670. Server 1100 then proceeds to step S1690.

Detailed Description Text (104):
In step S1690, if transmitter 1200 is already transmitting a packet over output link 1300 and is therefore not available to transmit an other packet, then server 1100 proceeds to step S1800. Otherwise, server 1100 proceeds to step S1700.

Detailed Description Text (105):
In step S1700, if the content of register 1120, which stores the value of the system potential, is equal to 0, then server 1100 proceeds to step S1780. Otherwise, server 1100 proceeds to step S1710.

Detailed Description Text (106):
In step S1710, server 1100 decrements the queue length register 1060-h for the connection h corresponding to the last packet being transmitted by transmitter 1200, and then proceeds to step S1720.

Detailed Description Text (107):
In step S1720, server 1100 proceeds to step S1722 if the content of queue length register 1060-h of connection h corresponding to the last packet being transmitted by transmitter 1200 is 0. Otherwise, server 1100 proceeds to step S1730.

Detailed Description Text (108):
In step S1722, server 1100 (more specifically, the controller 1130) decrements the content of register 1115. Server 1100 then proceeds to step S1724.

Detailed Description Text (109):
In step S1724, server 1100 (more specifically, the controller 1130) subtracts from the content of register 1110 the content of rate register 1040-h of connection h corresponding to the last packet being transmitted by transmitter 1200. Server 1100 then proceeds to step S1726.

Detailed Description Text (110):
In step S1726, server 1100 (more specifically, the controller 1130) subtracts the value at the head of packet length queue 1026-h of connection h corresponding to the last packet being transmitted by transmitter 1200 from the content of register 1117 and removes the value at the head of packet length queue 1026-h. Server 1100 then proceeds to step S1728.

Detailed Description Text (111):
In step S1728, server 1100 (more specifically, the controller 1130) subtracts the product of the contents of timestamp register 1050-h and rate register 1040-h of connection h corresponding to the last packet being transmitted by transmitter 1200 from the content of register 1123. Server 1100 then proceeds to step S1780.

Detailed Description Text (112):
In step S1730, server 1100 (more specifically, the controller 1130) subtracts the product of the contents of timestamp register 1050-h and rate register 1040-h of connection h corresponding to the last packet being transmitted by transmitter 1200 from the content of register 1123. Server 1100 then proceeds to step S1735.

Detailed Description Text (113):
In step S1735, server 1100 (more specifically, the controller 1130) subtracts the value at the head of packet length queue 1026-h of connection h corresponding to the last packet being transmitted by transmitter 1200 from the content of register 1117 and removes the value at the head of packet length queue 1026-h. Server 1100 then proceeds to step S1738.

Detailed Description Text (114):
In step S1738, server 1100 (more specifically, the controller 1140) determines the new timestamp of connection h corresponding to the last packet being transmitted by transmitter 1200 according to equation (5). Server 1100 then proceeds to step S1740.

Detailed Description Text (115):
In step S1740, server 1100 stores the value of the newly-generated timestamp of connection h corresponding to the last packet being transmitted by transmitter 1200 in timestamp register 1050-h, and then proceeds to step S1750.

Detailed Description Text (116):
In step S1750, server 1100 (more specifically, the controller 1130) adds the product of the contents of timestamp register 1050-h and rate register 1040-h of connection h corresponding to the last packet being transmitted by transmitter 1200 to the content of register 1123. Server 1100 then proceeds to step S1755.

Detailed Description Text (117):
In step S1755, server 1100 (more specifically, the controller 1130) adds the value at the head of packet length queue 1026-h of connection h corresponding to the last packet being transmitted by transmitter 1200 to the content of register 1117. Server 1100 then proceeds to step S1780.

Detailed Description Text (118):
In step S1780, the packet at the head of connection queue 1020-i corresponding to connection i identified in step S1680 by server 1100 (more specifically, by the selector 1170) is sent to transmitter 1200. Server 1100 then proceeds to step S1790.

Detailed Description Text (119):
In step S1790, server 1100 (more specifically, the controller 1135) determines the new value of the system potential according to equation (6), using the content of registers 1110, 1115, 1117, 1120, and 1123, and stores it in register 1120. Server 1100 also resets the content of counter 1127. Server 1100 then proceeds to step S1510.

Detailed Description Text (120):
In step S1800, if receiver 1010 has received new data packets, then server 1100 proceeds to step S1820. Otherwise, server 1100 proceeds to step S1690.

Detailed Description Text (121):
In step S1820, server 1100 selects one of the newly received data packets, and then proceeds to step S1830.

Detailed Description Text (122):
In step S1830, server 1100 identifies the connection w corresponding to the packet selected in step S1820, and then identifies connection w through a connection

identifier contained in the header of the packet (not shown). The identification of connection w allows to identify connection queue 1020-w corresponding to connection w, where the packet should be stored. Server 1100 then proceeds to step S1840.

Detailed Description Text (123):
In step S1840, server 1100 stores the packet selected in step S1820 in connection queue 1020-w, and then proceeds to step S1845.

Detailed Description Text (124):
In step S1845, server 1100 stores the length of the packet selected in step S1820 in packet length queue 1026-w and in register 1132, and then proceeds to step S1850.

Detailed Description Text (125):
In step S1850, server 1100 proceeds to step S1860 if the content of queue length register 1060-w is 0 (connection w is not backlogged). Otherwise, server 1100 proceeds to step S1940.

Detailed Description Text (126):
In step S1860, server 1100 (more specifically, the controller 1130) increments the content of register 1115, and then proceeds to step S1870.

Detailed Description Text (127):
In step S1870, server 1100 increments the content of queue length register 1060-w, and then proceeds to step S1880.

Detailed Description Text (128):
In step S1880, server 1100 (more specifically the controller 1135) determines the value of P(a.sub.i.sup.k), which appears in equation (4), according to equation (7), and provides the determined value to controller 1140. Server 1100 (more specifically, the controller 1140) then determines the new timestamp of connection w identified in step S1830 according to equation (4). Server 1100 then proceeds to step S1885.

Detailed Description Text (129):
In step S1885, server 1100 stores the timestamp of connection w in timestamp register 1050-w, and then proceeds to step S1890.

Detailed Description Text (130):
In step S1890, server 1100 (more specifically, the controller 1130) adds the content of rate register 1040-w to the content of register 1110. Server 1100 then proceeds to step S1900.

Detailed Description Text (131):
In step S1900, server 1100 (more specifically, the controller 1130) adds the content of register 1132 to the content of register 1117. Server 1100 then proceeds to step S1910.

Detailed Description Text (132):
In step S1910, server 1100 (more specifically, the controller 1130) adds the product of the contents of timestamp register 1050-w and rate register 1040-w to the content of register 1123. Server 1100 then proceeds to step S1930.

Detailed Description Text (133):
In step S1930, server 1100 proceeds to step S1820 if additional new data packets are available. Otherwise, server 1100 proceeds to step S1670.

Detailed Description Text (134):
In step S1940, server 1100 increments the content of queue length register 1060-w corresponding to connection w identified in step S1830, and then proceeds to step

S1930.

Detailed Description Text (135):
In step S1970, server 1100 increments the content of queue length register 1060-i corresponding to connection i identified in step S1560, and then proceeds to step S1975.

Other Reference Publication (1):
"Rate-Proportional Servers: A Design Methodology for Fair Queueing Algorithms," D. Stiliadis et al., Technical Report #UCSC-CLR-95-58, Dec. 1995, Baskin Center for Computer Engineering & Information Sciences, University of California, Santa Cruz, CA.

Other Reference Publication (2):
"Efficient Fair Queueng Algorithms for ATM and Packet Networks," D. Stiliadis et al., Technical Report #UCSC-CRL-95-59, Dec. 1995, Baskin Center for Computer Engineering & Information Sciences, University of California, Santa Cruz, CA.

Other Reference Publication (5):
"Carry-Over Round Robin: A Simple Cell Scheduling Mechanism for ATM Networks," D. Saha, et al., Proceedings IEEE INFOCOM '96 on Computer Communications, Mar. 24-28, 1996, vol. 2, p. 630-637.

CLAIMS:

1. A method of servicing, at a predetermined service rate, a plurality of queues containing data packets, each of said queues being associated with respective connections, said connections traversing an associated communications switch, each of said connections being allocated a respective data transfer rate, said method comprising:

responsive to receiving a plurality of data packets via a plurality of data links, identifying for each received data packet the respective one of said connections and identifying the associated one of said queues,

storing each of the received data packets in their respective identified queue,

associating a timestamp with each connection whose associated queue has at least one data packet waiting therein, in which said connection is identified as a backlogged connection, and generating a timestamp associated with each connection each time a new data packet reaches the head of the associated queue,

storing in memory the timestamps associated with respective ones of the backlogged connections,

determining which of the timestamps associated with respective ones of the backlogged connections has the smallest value among all of the timestamps, identifying the associated connection, removing a data packet from the head of that one of the queues associated with the identified connection and transmitting the removed data packet to an output,

in which the timestamp associated with a backlogged connection, generated each time a new data packet reaches the head of the associated queue, is generated as a maximum value between a previous value of the timestamp assigned to the connection and a current value of at least one function or system potential, said maximum value incremented by the inverse of the data transfer rate allocated to the connection normalized to the rate of the server, if the connection associated with the data packet was not backlogged before the packet reaches the head of the associated queue,

in which the timestamp associated with a backlogged connection, generated each time a new data packet reaches the head of the associated queue, is generated as the previous value of the timestamp of the connection incremented by the inverse of the data transfer rate allocated to the connection normalized to the rate of the server, if the connection associated with the data packet was backlogged before the packet arrived at the head of the associated queue,

wherein a value for the system potential is generated following the transmission of a packet in the system as the maximum value between the latest value of the system potential incremented by a first predetermined value and the weighted sum of the timestamp values associated with all the currently backlogged connections, in which the weight of each timestamp value in the weighted sum is the data transfer rate allocated to the associated connection, and in which said weighted sum is decremented by the number of connections that are currently backlogged, in which that result is then divided by the sum of the values of the data transfer rates allocated to all connections that are currently backlogged.

2. The method of claim 1 further comprising the step of

responsive to when there are no data packets waiting in the plurality of queues, resetting to zero the value of the system potential and the values of the timestamps associated with respective ones of all the connections.

3. The method of claim 1 further comprising the step of

responsive to when there are no data packets waiting in the plurality of queues, setting the value of the system potential equal to or larger than the largest of the timestamp values associated with the connections.

4. The method of claim 1 further comprising the step of

following the transmission of a packet in the system, determining the value of the at least one function or system potential as a maximum value between the latest value of the system potential incremented by the first predetermined value and the value of the timestamp associated with the connection served most recently by the server, said value of the timestamp associated with the connection served most recently by the server decremented by the number of all the connections that are currently backlogged and then divided by the sum of the values of the data transfer rates allocated to all connections that are currently backlogged.

5. The method of claim 1, further comprising:

determining the length of each received data packet and storing the determined length in a memory location associated with the queue associated with the data packet,

in which the timestamp associated with a backlogged connection, generated each time a new data packet reaches the head of the associated queue, is generated as the maximum value between the previous value of the timestamp assigned to the connection and the current value of at least one function or system potential, said maximum value incremented by the length of the data packet divided by the data transfer rate allocated to the connection normalized to the rate of the server, if the connection associated with the data packet was not backlogged before the packet arrived at the head of the associated queue,

in which the timestamp associated with a backlogged connection, generated each time a new data packet reaches the head of the associated queue, is generated as the value of the previous value of the timestamp of the connection incremented by the length of the data packet that has just reached the head of the associated queue divided by the data transfer rate allocated to the connection normalized to the

rate of the server, if the connection associated with the data packet was backlogged before the packet arrived at the head of the associated queue,

wherein a value for the system potential is generated following the transmission of a packet in the system as a maximum value between the latest value of the system potential incremented by a first duration factor, in which the duration factor is equal to the duration of the period of time that starts from the time when the previous value of the system potential was generated and ends at the current time when the value of the system potential is generated, and the weighted sum of the timestamp values associated with the currently backlogged connections, in which the weight of each timestamp value in the weighted sum is the allocated data transfer rate of the associated connection, and in which said weighted sum is decremented by the sum of the lengths of the data packets at the head of all queues associated with connections that are currently backlogged and then divided by the sum of the values of the data transfer rates allocated to all connections that are currently backlogged,

wherein a value for the system potential is generated following the receiver receiving a packet associated with a connection that was not backlogged prior of such arrival as the latest value of the system potential incremented by the first duration factor.

6. The method of claim 5 further comprising the step of

responsive to when there are no data packets waiting in the plurality of queues, resetting to zero the value of the system potential and the values of the timestamps associated with respective ones of all the connections.

7. The method of claim 5 further comprising the step of p1 responsive to when there are no data packets waiting in the plurality of queues, setting the value of the system potential equal to or larger than the largest of the timestamp values associated with the connections.

8. The method of claim 5 further comprising the step of p1 following the transmission of a packet in the system, determining the value of the at least one function or system potential as the maximum value between the latest value of the at least one function or system potential incremented by the first duration factor and the value of the timestamp associated with the connection served most recently by the server, said value of the timestamp associated with the connection served most recently by the server decremented by the sum of the lengths of the data packets at the head of all queues associated with connections that are currently backlogged and then divided by the sum of the values of the data transfer rates allocated to all connections that are currently backlogged.

9. An apparatus for servicing, at a predetermined service rate, a plurality of queues containing data packets, said queues being associated with respective connections, said connections traversing an associated communications switch, each of said connections being allocated a respective data transfer rate, said apparatus comprising:

memory forming the plurality of queues associated with respective ones of said connections,

a receiver for receiving a plurality of data packets via a plurality of data links, for identifying for each received data packet the respective one of said connections, and identifying the associated one of said queues and for storing each of the received data packets in their respective identified queue,

a first controller for associating a timestamp with each connection whose associated queue has at least one data packet waiting therein, in which said

connection is identified as a backlogged connection, and for generating a timestamp associated with each connection each time a new data packet reaches the head of the associated queue, and for storing in memory the timestamps associated with respective ones of the backlogged connections,

a second controller for determining which of the timestamps associated with respective ones of the backlogged connections has the smallest value among all of the timestamps, and for identifying the associated connection, and for removing a data packet from the head of that one of the queues associated with the identified connection and transmitting the removed data packet to an output,

said first controller including apparatus, operative when the connection associated with a data packet that has just reached the head of the associated queue was not backlogged before that packet reached the head of the associated queue, for then generating the timestamp associated with that backlogged connection as a maximum value between the previous value of the timestamp assigned to the connection and the current value of at least one function or system potential, said maximum value incremented by the inverse of the data transfer rate allocated to the connection normalized to the rate of the server, and

operative when the connection associated with a data packet that has just reached the head of the associated queue was backlogged before that packet reached the head of the associated queue, for then generating the timestamp associated with that backlogged connection as the previous value of the timestamp of the connection incremented by the inverse of the data transfer rate allocated to the connection normalized to the rate of the server,

operative following the transmission of a packet in the system for generating a value for the system potential as the maximum value between the latest value of the system potential incremented by a first predetermined value and the weighted sum of the timestamp values associated with all the currently backlogged connections, in which the weight of each timestamp value in the weighted sum is the data transfer rate allocated to the associated connection, and in which said weighted sum is decremented by the number of connections that are currently backlogged, in which that result is then divided by the sum of the values of the data transfer rates allocated to all connections that are currently backlogged.

10. The apparatus of claim 9 wherein said first controller further includes apparatus, responsive to when there are no data packets waiting in the plurality of queues, for then resetting to zero the value of the system potential and the values of the timestamps associated with respective ones of all the connections.

11. The apparatus of claim 9 wherein said first controller further includes apparatus, responsive to when there are no data packets waiting in the plurality of queues, for then setting the value of the system potential equal to or larger than the largest of the timestamp values associated with the connections.

12. The apparatus of claim 9 wherein said first controller further includes apparatus, operative following the transmission of a packet in the system, for determining the value of the at least one function or system potential as the maximum between the latest value of the system potential incremented by the first predetermined value and the value of the timestamp associated with the connection served most recently by the server, said value of the timestamp associated with the connection served most recently by the server decremented by the number of all the connections that are currently backlogged and then divided by the sum of the values of the data transfer rates allocated to all connections that are currently backlogged.

13. The apparatus of claim 9, wherein the receiver further includes apparatus for determining the length of each received data packet and storing the determined

length in a memory location associated with the queue associated with the data packet,

and the first controller including apparatus, operative when the connection associated with a data packet that has just reached the head of the associated queue was not backlogged before that packet reached the head of the associated queue, for then generating the timestamp associated with that backlogged connection as the maximum value between the previous value of the timestamp assigned to the connection and the current value of at least one function or system potential, said maximum value incremented by the length of the data packet divided by the data transfer rate allocated to the connection normalized to the rate of the server,

operative when the connection associated with a data packet that has just reached the head of the associated queue was backlogged before that packet reached the head of the associated queue, for then generating the timestamp associated with that backlogged connection as the previous value of the timestamp of the connection incremented by the length of the data packet that has just reached the head of the associated queue divided by the data transfer rate allocated to the connection normalized to the rate of the server,

operating following the transmission of a packet in the system for generating a value for the system potential as the maximum value between the latest value of the system potential incremented by a first duration factor, in which the duration factor is equal to the duration of the period of time that starts from the time when the previous value of the system potential was generated and ends at the current time when the value of the system potential is generated, and the weighted sum of the timestamp values associated with the currently backlogged connections, in which the weight of each timestamp value in the weighted sum is the allocated data transfer rate of the associated connection, and in which said weighted sum is decremented by the sum of the lengths of the data packets at the head of all queues associated with connections that are currently backlogged and then divided by the sum of the values of the data transfer rates allocated to all connections that are currently backlogged,

operating following the receiver receiving a packet associated with a connection that was not backlogged prior of such arrival for generating a value for the system potential as the latest value of the system potential incremented by the first duration factor.

14. The apparatus of claim 13 wherein said first controller further includes apparatus, responsive to when there are no data packets waiting in the plurality of queues, for then resetting to zero the value of the system potential and the values of the timestamps associated with respective ones of all the connections.

15. The apparatus of claim 13 wherein said first controller further includes apparatus, responsive to when there are no data packets waiting in the plurality of queues, for then setting the value of the system potential equal to or larger than the largest of the timestamp values associated with the connections.

16. The apparatus of claim 13 wherein said first controller further includes apparatus, operative following the transmission of a packet in the system, for determining the value of the at least one function or system potential as the maximum value between the latest value of the at least one function or system potential incremented by the first duration factor and the value of the timestamp associated with the connection served most recently by the server, said value of the timestamp associated with the connection served most recently by the server decremented by the sum of the lengths of the data packets at the head of all queues associated with connections that are currently backlogged and then divided by the sum of the values of the data transfer rates allocated to all connections that are currently backlogged.

17. A method of servicing a plurality of queues containing data packets at a predetermined rate, said queues being associated with respective connections through an associated communications switch, each of said connections being associated with a respective data transfer rate, said method comprising:

responsive to receiving a plurality of data packets via a plurality of data links, identifying for each received data packet the respective one of said connections and identifying the associated one of the queues as a function of the identified connection,

storing each of the received data packets in their respective identified queues,

generating a timestamp for each connection whose associated queue has least one data packet stored therein and the data packet has reached the head of that queue, in which a connection associated with such a queue is identified as a backlogged connection,

storing in memory the timestamps in association with respective ones of the backlogged connections,

determining which of the time stamps associated with respective ones of the backlogged connections has the smallest value, removing a data packet from the head of that one of the queues associated with the determined connection and transmitting the removed data packet to an output,

if the determined connection was not backlogged before the packet reached the head of the associated queue, then generating the timestamp for that queue as a maximum value within a range of values extending from the previous value of the timestamp generated for the associated connection and to a current value of a system potential, in which the maximum value is incremented by the inverse of the data transfer rate allocated to the associated connection normalized to the service rate,

if the determined connection was backlogged before the packet reached the head of the associated queue, then generating the timestamp when a new data packet arrives at the head of a queue associated with a backlogged connection as a function of the value of the previous timestamp generated for that connection incremented by the inverse of the data transfer rate allocated to the connection normalized to the service rate, and

following the transmission of a packet in the system, generating a value for the system potential as a maximum value within a range extending from the latest value of the system potential incremented by a first predetermined value and a weighted sum of the timestamp values associated with all of the currently backlogged connections, in which the weight of each timestamp value in the weighted sum is the data transfer rate allocated to the associated connection, and in which the weighted sum is decremented by the number of connections that are currently backlogged, in which that result is then divided by the sum of the values of the data transfer rates allocated to all connections that are currently backlogged.

18. The method of claim 17 further comprising the step of

responsive to no data packets waiting in the plurality of queues, setting to zero the value of the system potential and the values of the timestamps associated with respective ones of the connections.

19. The method of claim 17 further comprising the step of

responsive to no data packets waiting in the plurality of queues, setting the value

of the system potential equal to or larger than the largest of the timestamp values.

21. The method of claim 17 further comprising the step of

determining the length of each received data packet and storing the determined length in a memory location associated with the identified connection,

determining, each time a data packet stored in a queue associated with a backlogged connection reaches the head of that queue, a timestamp as a maximum value ranging between the previous value of the timestamp assigned to the backlogged connection and the current value of the system potential, in which the maximum value is incremented by the length of the data packet divided by the data transfer rate allocated to the associated connection normalized to the service rate, if associated connection was not backlogged before the packet reached the head of the associated queue,

when a data packet reaches the head of a queue associated with a backlogged connection and if that connection was backlogged before the packet reached the head of the associated queue, generating a timestamp as the value of the previous value of the timestamp for that connection incremented by the length of the latter data packet divided by the data transfer rate allocated to the associated backlogged connection normalized to the service rate of the server, and

following the transmission of data packet, determining the system potential system as a maximum value in a range of values extending from the current value of the system potential incremented by the first predetermined value and the weighted sum of the timestamp values respectively associated with the currently backlogged connections, in which the weight of each timestamp value in the weighted sum is the data transfer rate allocated to the associated connection, and in which the weighted sum is decremented by the sum of the lengths of the data packets at the head of queues respectively associated with connections that are currently backlogged and then divided by the sum of the values of the data transfer rates allocated to all such connections.

22. The method of claim 21 further comprising the step of

responsive to when there are no data packets waiting in the plurality of queues, resetting the value of the system potential and the value of all the timestamps to zero.

23. The method of claim 21 further comprising the step of

responsive to when there are no data packets waiting in the plurality of queues, setting the value of the system potential equal to or larger than the value of the timestamp having the largest value.

24. The method of claim 21 further comprising the step of

following the transmission of a packet, determining the value of the system potential as a maximum value in a range of values extending from the current value of the system potential incremented by the predetermined value and the value of the timestamp associated with the connection currently being served by the system, in which the latter timestamp is decremented by the sum of the lengths of the data packets that have reached the head of queues associated with connections that are currently backlogged and then divided by the sum of the values of the data transfer rates allocated to all such connections.

25. Apparatus for servicing a plurality of queues containing received data packets at an predetermined rate, said queues being associated with respective connections

established through an associated communications switch, each of said connections being associated with a respective data transfer rate, said apparatus comprising:

memory forming a plurality of queues associated with respective ones of said connections,

a receiver for receiving data packets via a plurality of data links, and identifying for each received data packet the associated one of said connections and identifying the associated one of the queues as a function of the identified connection and for storing each of the received data packets in their respective identified queues,

a first controller for generating a timestamp for each connection whose associated queue has least one data packet stored therein and a data packet priorly stored in the associated queue has reached the head of that queue, in which a connection associated with such a queue is identified as a backlogged connection, and for storing the timestamps in respective memory locations as they are generated,

a second controller for determining which one of the time stamps associated with respective ones of the backlogged connections has the smallest value among all of the time stamps, for removing a data packet from the head of that one of the queues associated with the determined connection and transmitting the removed data packet to an output,

said second controller including apparatus, operative if the determined connection was not backlogged before the packet reached the head of the associated queue, for then generating the timestamp for that queue as a maximum value within a range of values extending from the previous value of the timestamp generated for the associated connection to a current value of a system potential, in which the maximum value is incremented by an inverse of the data transfer rate allocated to the associated connection normalized to the service rate,

operative if the determined connection was backlogged before the packet reached the head of the associated queue, for then generating the timestamp when a new data packet arrives at the head of a queue associated with a backlogged connection as a function of the value of the previous timestamp generated for that connection incremented by the inverse of the data transfer rate allocated to the connection normalized to the service rate, and

operative, following the transmission of a packet in the system, for generating a value for the system potential as a maximum value within a range extending from the latest value of the system potential incremented by a first predetermined value and a weighted sum of the timestamp values associated with all of the currently backlogged connections, in which the weight of each timestamp value in the weighted sum is the data transfer rate allocated to the associated connection, and in which the weighted sum is decremented by the number of connections that are currently backlogged, in which that result is then divided by the sum of the values of the data transfer rates allocated to all connections that are currently backlogged.

26. The apparatus of claim 25 wherein said second controller further includes apparatus, responsive to no data packets waiting in the plurality of queues, for then setting to zero the value of the system potential and the values of the timestamps associated with respective ones of the connections.

27. The apparatus of claim 25 wherein said second controller further includes apparatus, responsive to no data packets waiting in the plurality of queues, for then setting the value of the system potential equal to or larger than the largest of the timestamp values.

29. The method of claim 27 wherein said second controller further includes

apparatus for

determining the length of each received data packet and storing the determined length in a memory location associated with the identified connection, and apparatus

operative each time a data packet stored in a queue associated with a backlogged connection reaches the head of that queue and if the associated connection was not backlogged before the packet reached the head of the associated queue, for determining a timestamp as a maximum value ranging between the previous value of the timestamp assigned to the backlogged connection and the current value of the system potential, in which the maximum value is incremented by the length of the data packet divided by the data transfer rate allocated to the associated connection normalized to the service rate,

operative when a data packet reaches the head of a queue associated with a backlogged connection and if that connection was backlogged before the packet reached the head of the associated queue, for generating a timestamp as the value of the previous value of the timestamp for that connection incremented by the length of the latter data packet divided by the data transfer rate allocated to the associated backlogged connection normalized to the service rate of the server, and

operative following the transmission of data packet, for determining the system potential system as a maximum value in a range of values extending from the current value of the system potential incremented by the first predetermined value and the weighted sum of the timestamp values respectively associated with the currently backlogged connections, in which the weight of each timestamp value in the weighted sum is the data transfer rate allocated to the associated connection, and in which the weighted sum is decremented by the sum of the lengths of the data packets at the head of queues respectively associated with connections that are currently backlogged and then divided by the sum of the values of the data transfer rates allocated to all such connections.

30. The apparatus of claim 29 wherein said second controller further includes apparatus, responsive to when there are no data packets waiting in the plurality of queues, for resetting the value of the system potential and the value of all the timestamps to zero.

31. The apparatus of claim 29 wherein said second controller further includes apparatus, responsive to when there are no data packets waiting in the plurality of queues, for setting the value of the system potential equal to or larger than the value of the timestamp having the largest value.

32. The apparatus of claim 29 wherein said second controller further includes apparatus operative, following the transmission of a packet, for determining the value of the system potential as a maximum value in a range of values extending from the current value of the system potential incremented by the predetermined value and the value of the timestamp associated with the connection currently being served by the system, in which the latter timestamp is decremented by the sum of the lengths of the data packets that have reached the head of queues associated with connections that are currently backlogged and then divided by the sum of the values of the data transfer rates allocated to all such connections.

First Hit Fwd Refs

(9) 1

☐ Generate Collection Print

L19: Entry 5 of 9 File: USPT Oct 2, 2001

DOCUMENT-IDENTIFIER: US 6298386 B1
TITLE: Network file server having a message collector queue for connection and connectionless oriented protocols

Abstract Text (1):
There is a performance loss associated with servicing a pipe or stream for a connection oriented process by maintaining a connection between a server thread and a client for a series of messages. As a result of maintaining this connection, there is less balance; some threads work harder than others, causing a loss of performance. To solve this problem, a collector queue combines messages from the connection oriented process with messages from the other concurrent processes. The threads receive messages from the collector queue rather than individual pipes. Any idle thread can pick up a message from the collector queue. The collector queue keeps track of which pipe each message came from so that the reply of the server to each message is directed to the same pipe from which the message came from. Therefore the collector queue ensures thread balance and efficiency in servicing the messages. In the preferred implementation, each entry in the collector queue includes a message pointer and a pipe pointer. The message pointer points to allocated memory storing the message in a message buffer. The pipe pointer points to the pipe from which the message originated. The collector queue is a singly linked list. A free thread takes an entry off the collector queue, interprets the message of the entry, sends a reply, and deallocates the memory of the entry and the allocated memory storing the message in the message buffer.

Parent Case Text (2):
The present application is a continuation-in-part of provisional application Ser. No. 60/023,914 filed Aug. 14, 1996, which is incorporated herein by reference, and has the following additional continuation-in-part applications: Percy Tzelnic et al., Ser. No. 08/747,875 filed Nov. 13, 1996, entitled "Network File Server Using an Integrated Cached Disk Array and Data Mover Computers"; Percy Tzelnic et al., Ser. No. 08/748,363 filed Nov. 13, 1996, entitled "Network File Server Maintaining Local Caches of File Directory Information in Data Mover Computers"; and Uresh K. Vahalia et al., Ser. No. 08/747,631 filed Nov. 13, 1996, entitled "File Server Having a File System Cache and Protocol for Truly Safe Asynchronous Writes."

Parent Case Text (3):
Percy Tzelnic et al., Ser. No. 08/747,875 filed Nov. 13, 1996, entitled "Network File Server Using an Integrated Cached Disk Array and Data Mover Computers," is a contiunation-in-part of provisional application Ser. No. 60/005,988 filed Oct. 27, 1995 by Percy Tzelnic et al., entitled "Video File Server," incorporated herein by reference, and its pending divisional applications: Percy Tzelnic et al., Ser. No. 08/661,152 filed Jun. 10, 1996, entitled "Video File Server Using an Integrated Cached Disk Array and Stream Server Computers; Natan Vishlitzky et al., Ser. No. 08/661,185 filed Jun. 10, 1996, entitled "Prefetching to Service Multiple Video Streams from an Integrated Cached Disk Array", issued on Apr. 7, 1998 as U.S. Pat. No. 5,737,747; Uresh Vahalia et al., Ser. No. 08/661,053 filed Jun. 10, 1996, entitled "Staggered Stream Support for Video On Demand"; and Percy Tzelnic et al., Ser. No. 08/661,187 filed Jun. 10, 1996, entitled "On-Line Tape Backup Using an Integrated Cached Disk Array," issued on Oct. 27, 1998 as U.S. Pat. No. 5,829,046; which are all incorporated herein by reference.

Brief Summary Text (3):
The present invention relates generally to a network file server, and more particularly to a network file server servicing a number of clients simultaneously.

Brief Summary Text (5):
A network file server may support any number of client-server communication protocols, such as the User Datagram Protocol (UDP) and the Transmission Control Protocol (TCP).

Brief Summary Text (6):
UDP is a connectionless protocol. There is one fast pipe or stream of messages (e.g., requests) over a network link from a number of clients to one or more servers. The messages get mixed together during transmission in the pipe.

Brief Summary Text (7):
TCP is a connection oriented protocol. Each client is assigned a separate pipe to a respective server.

Brief Summary Text (8):
The servers execute code threads that service respective client messages. In the UDP case, there are lots of code threads to service client messages.

Brief Summary Text (9):
In the TCP case, the threads are assigned to respective pipes, and the pipes are assigned to respective clients. Therefore, the threads remain connected to the respective clients. There are fewer TCP threads, and some threads are very busy and others are not very busy, since the threads remain connected to the clients.

Brief Summary Text (11):
The inventors have recognized that there is a performance loss associated with servicing a TCP pipe by maintaining a connection between a server thread and a client for a series of messages. As a result of maintaining this connection, there is less balance; some threads work harder than others, causing a loss of performance. This occurs in Network File System (NFS) servers, such as NFS servers from Sun Microsystems Inc. and Digital Equipment Corp.

Brief Summary Text (12):
In accordance with a basic aspect of the invention, a collector queue minimizes the loss of performance due to thread imbalance when servicing messages from a connection oriented process concurrent with other processes. The collector queue combines messages from the connection oriented process with messages from the other concurrent processes. The other concurrent processes may include connection oriented processes such as TCP and connectionless processes such as UDP. The threads of the server receive messages from the collector queue rather than individual pipes. Any idle thread can pick up a message from the collector queue. The collector queue keeps track of which pipe each message came from so that the reply of the server to each message is directed to the same pipe from which the message came from. Therefore the collector queue ensures thread balance and efficiency in servicing the messages.

Brief Summary Text (13):
In the preferred implementation, each entry in the collector queue includes a message pointer and a pipe pointer. The message pointer points to allocated memory storing the message in a message buffer. The pipe pointer points to the pipe from which the message originated. The collector queue is a singly linked list. There is a pool of threads, and a free thread takes an entry off the collector queue, interprets the message of the entry, sends a reply, and deallocates the memory of the entry and the allocated memory storing the message in the message buffer.

h e b b g e e e f c e e ge

Drawing Description Text (3):
FIG. 1 is a perspective view of a network file server that incorporates the present invention;

Drawing Description Text (4):
FIG. 2 is a block diagram of the network file server of FIG. 1 and its connections to a network;

Drawing Description Text (5):
FIG. 3 is a block diagram of an integrated cached disk array storage subsystem used in the network file server of FIG. 1;

Drawing Description Text (6):
FIG. 4 is a block diagram showing software structure in the network file server of FIG. 1;

Drawing Description Text (9):
FIG. 7 is a block diagram showing caching, exchange, and replication of file directory and locking information among data mover computers in the network file server of FIG. 1;

Drawing Description Text (10):
FIG. 8 is a first portion of a flowchart illustrating a file manager program in a data mover computer that caches, exchanges, and replicates file directory and locking information among the data mover computers during a file access task in the network file server of FIG. 1;

Drawing Description Text (15):
FIG. 13, labeled "Prior Art," is a block diagram of a conventional UNIX server;

Drawing Description Text (16):
FIG. 14 is a block diagram of a UNIX server that has been modified to solve an asynchronous write security problem;

Drawing Description Text (17):
FIG. 15 is a flowchart of programming in a file system layer of the modified UNIX server of FIG. 14;

Drawing Description Text (20):
FIG. 18 is a block diagram showing the use of a collector queue combining UDP messages with TCP messages and permitting a next message in the collector queue to be serviced by an idle code thread implementing file access protocols in a server;

Drawing Description Text (22):
FIG. 20 is a flowchart of programming for a code thread that services a next message in the collector queue; and

Detailed Description Text (2):
I. The Architecture of a Network File Server

Detailed Description Text (3):
Turning now to FIG. 1 of the drawings, there is shown a network file server generally designated 20 incorporating the present invention. The network file server 20 includes an array of data movers 21, a network server display and keyboard 32, an integrated cached disk array storage subsystem (ICDA) 23, and an optional tape silo 24. At least two of the data movers 28, 29 are also programmed to service the network server display and keyboard 32, and these particular data movers will be referred to as display and keyboard servers. However, at any given time, only one of the display and keyboard servers 28, 29 is active in servicing

the network server display and keyboard 32.

Detailed Description Text (4):
The network file server 20 is managed as a dedicated network appliance, integrated with popular network operating systems in a way, which, other than its superior performance, is transparent to the end user. It can also be provided with specialized support for isochronous data streams used in live, as well as store-and-forward, audio-visual applications, as described in the above-referenced Percy Tzelnic et al. provisional application Ser. No. 60/005,988 entitled "Video File Server," and its divisional applications: Percy Tzelnic et al., Ser. No. 08/661,152 filed Jun. 10, 1996, entitled "Video File Server Using an Integrated Cached Disk Array and Stream Server Computers; Natan Vishlitzky et al., Ser. No. 08/661,185 filed Jun. 10, 1996, entitled "Prefetching to Service Multiple Video Streams from an Integrated Cached Disk Array"; Uresh Vahalia et al., Ser. No. 08/661,053 filed Jun. 10, 1996, entitled "Staggered Stream Support for Video On Demand"; and Percy Tzelnic et al., Ser. No. 08/661,187 filed Jun. 10, 1996, entitled "On-Line Tape Backup Using an Integrated Cached Disk Array;" which are all incorporated herein by reference.

Detailed Description Text (5):
The network file server 20 is directed to high-end file server applications such as the Network File System (NFS, version 2 and 3) (and/or other access protocols). NFS is a well-known IETF file access protocol standard (RFC 1094, Sun Microsystems, Inc., "NFS: Network File System Protocol Specification," Mar. 1, 1989). NFS acts as a network server for network communications by providing basic file access operations for network clients. Such basic file access operations include opening a file, reading a file, writing to a file, and closing a file.

Detailed Description Text (7):
Each of the data movers 21, including the display and keyboard servers 28, 29, is a high-end commodity computer, providing the highest performance appropriate for a data mover at the lowest cost. The data movers 21 are mounted in a standard 19" wide rack. Each of the data movers 21, for example, includes an Intel processor connected to a EISA or PCI bus and at least 64 MB of random-access memory. The number of the data movers 21, their processor class (i486, Pentium, etc.) and the amount of random-access memory in each of the data movers, are selected for desired performance and capacity characteristics, such as the number of concurrent network clients to be serviced. Preferably, one or more of the data movers 21 are kept in a standby mode, to be used as "hot spares" or replacements for any one of the other data movers that fails to acknowledge commands from the other data movers or is otherwise found to experience a failure.

Detailed Description Text (8):
Each of the data movers 21 contains one or more high-performance FWD (fast, wide, differential) SCSI connections to the ICDA 23. Each of the data movers 21 may also contain one or more SCSI connections to the optional tape silo 24. Each of the data movers 21 also contains one or more bidirectional network attachments 30 configured on the data mover's EISA or PCI bus. The network attachments 30, for example, are Ethernet, FDDI, ATM, DS1, DS3, or channelized T3 attachments to data links to a network (25 in FIG. 2). The network 25 connects these network attachments to the network clients 54, for example, through an ATM switch 53. Each of the data movers 21 also includes an additional Ethernet connection to an internal dual-redundant Ethernet link (26 in FIG. 2) for coordination of the data movers with each other, including the display and keyboard servers 28, 29.

Detailed Description Text (9):
The display and keyboard server 28, 29 active for servicing of the display and keyboard 32 can also conduct one or more standard management and control protocols such as SNMP (RFC 1157, M. Schoffstall, M. Fedor, J. Davin, J. Case, "A Simple Network Management Protocol (SNMP)," May 10, 1990). SNMP is an internet protocol

that permits inspection and modification of system variables such as the network address (IP) and the number of buffers for network communication. In addition to the connections described above that the data movers 21 have to the network 25, the cached disk array 23, and the optional tape silo 24, each of the display and keyboard servers 28, 29 also has a connection to a serial link 31 to the network server display and keyboard 32. The display and keyboard servers 28, 29 run a conventional operating system (such as Windows NT or UNIX) to provide a hot-failover redundant configuration for servicing of the display and keyboard 32. An operator at the display and keyboard 32 uses SNMP for management and control of the resources of the network file server 20.

Detailed Description Text (11):
Turning now to FIG. 2, there is shown a block diagram of the network file server 20 including the SCSI connections 40 among the cached disk array 23, the optional tape silo 24, the controller servers 28, 29, and the data movers 21. The cached disk array 23 includes a large capacity semiconductor cache memory 41 and SCSI adapters 45 providing one or more FWD SCSI links to each of the data movers 21, including the display and keyboard servers 28, 29.

Detailed Description Text (12):
The optional tape silo 24 includes an array of SCSI adapters 50 and an array of read/write stations 51. Each of the read/write stations 51 is connected via a respective one of the SCSI adapters 50 and a FWD SCSI link to a respective one of the data movers 21, including the display and keyboard servers 28, 29. The read/write stations 51 are controlled robotically in response to commands from the data movers 21 for tape transport functions, and preferably also for mounting and unmounting of tape cartridges into the read/write stations from storage bins.

Detailed Description Text (13):
In a preferred mode of operation, to archive data from a file from the network to tape, one of the data movers 21 receives the file from the network 25 and prestages the file to the cached disk array 23 at a high rate limited by the network transmission rate (about 150 GB/hour). Then one of the data movers 21 destages the file from the cached disk array 23 to an associated one of the read/write stations 51 at a tape device speed (about 7 GB/hour). For most applications, prestaging to disk can be done immediately, and staging from disk to tape including sorting of files onto respective tape cassettes can be done as a background operation or at night, when the load on the network file server 20 is at a minimum. In this fashion, the cached disk array 23 can absorb a high data inflow aggregation from tens or hundreds of network links streaming from multiple sites, and balance this load on the read/write stations 41. Prestaging to the integrated cached disk array allows better use of the read/write stations 51, matching of server flow to tape streaming flow, and reduction of tape and read/write station wear. Prestaging to the back-end also allows multiple classes of backup and restore services, including instant backup for files maintained on disk in the cached disk array 23, and temporary batch backup pending a success or failure acknowledgment. Prestaging to the cached disk array 23 also makes economical an on-line archive service performing the staging from the cached disk array 23 to tape as a background process.

Detailed Description Text (16):
II. Network File Server Software

Detailed Description Text (17):
Turning now to FIG. 4, there is shown a block diagram of software 60 providing a real-time processing environment in the network file server (20 of FIGS. 1 and 2). The software 60 is executed by the processors of the data movers 21, including the display and keyboard servers 28, 29. The software 60 also provides an environment for managing file services and multiple high-performance data streams as well as a standard set of service-level application program interfaces (APIs) for developing

and porting file service protocols (such as NFS). The software 60 is an application run by a general purpose operating system such as Microsoft NT.

Detailed Description Text (20):
The software 60 further includes an SNMP management agent 64 supporting a Simple Network Management Protocol. SNMP is a standard internet protocol for inspecting and changing system variables. For example, the SNMP management agent is used when an operator at the network server display and keyboard (32 in FIG. 1) sets the network IP address of the network file server (20 in FIG. 1).

Detailed Description Text (22):
The group of software modules providing communication between the common file system and the network includes file access protocols 75 and a network server interface 73 using communication stacks 74 and network link drivers 72. The file access protocols 75 include a set of industry standard network server protocols such as NFS. Other file access protocols compatible with the network 25 could also be used, such as Novell NCP, LanManager, SMB, etc.

Detailed Description Text (23):
The file access protocols 75 are layered between the communication stacks 74 and the common file system 71. The communication stacks 74 provide network access and connectivity for the data transmitted to the file access protocol layer 75 from the network link drivers 72. The communication stacks include TCP/IP, IPX/SPX, NETbeui, or others. The network server interface 73 allows porting of the network software and file access protocols 72, 74, 75. This interface 73 is System V Streams. There could be multiple concurrent instances of the file access protocols 75, communication stacks 74, and drivers 72.

Detailed Description Text (26):
The file server software runs as an embedded system that includes a real-time kernel (63 in FIGS. 4 and 5). The main components of the kernel are a task scheduler, frameworks for writing device drivers, and a number of system services that are commonly found in similar real-time kernels. The system services include kernel interfaces to memory management, timers, synchronization, and task creation. All kernel tasks run in a single unprotected address space. As a result of this, no copy operations are required to move data from the cached disk array 23 to the network. Copying is eliminated by passing references to common buffers across all subsystems.

Detailed Description Text (29):
The general-purpose class is implemented as a standard threads package, with a thread corresponding to a general-purpose task as described herein. A suitable threads package is described in A. D. Birrell, "An Introduction to Programming with Threads," Systems Research Center Technical Report, No. 35, Digital Equipment Corporation, Maynard, Mass., (1989).

Detailed Description Text (31):
In the network file server, real-time tasks are used to implement "polling" device drivers and communication stacks. The method of polling for pending work, as opposed to interrupt-driven processing, contributes to system stability and alleviates most of the problems that arise during overloads. It also provides isolation between multiple real-time tasks that have differing performance requirements. Polling regulates the flow of traffic into the network file server. Just as flow control mechanisms, such as a leaky bucket scheme, protect network resources from large bursts, polling protects the end-system resources by regulating the frequency at which work queues are scanned and limiting the amount of work that may be performed during each scan of the round-robin schedule.

Detailed Description Text (37):
There are two basic objectives in organizing the respective tasks of the cached

disk array 23 and the data movers 21 in the network file server 20 of FIG. 1. The first and primary objective is to organize the respective tasks so that the processing load on the cached disk array 23 is balanced with the processing load on the data movers 21. This balancing ensures that neither the cached disk array 23 nor the data movers 21 will be a bottleneck to file access performance. The second basic objective is to minimize modifications or enhancements to the cached disk array 23 to support network file access.

Detailed Description Text (38):
To some degree, the second objective is driven by a desire to minimize marketing and support issues that would arise if the cached disk array 23 were modified to support network file access. The second objective is also driven by a desire to minimize the addition of processing load on the cached disk array associated with network file access. The network file server architecture of FIG. 1 permits data mover computers 21 to be added easily until the cached disk array 23 becomes a bottleneck to file access performance, and therefore any additional processing load on the cached disk array associated with network file access would tend to cause a reduction in the network file access performance of a fully configured system employing a single cached disk array.

Detailed Description Text (48):
Each data mover, such as the data movers 21a and 21b shown in FIG. 7, includes a local directory (94a, 94b) of locking information for all locked network files accessible by the data mover. Each local directory (94a, 94b) of locking information for locked network files includes a file to logical block mapping (95a, 95b), file attributes (96a, 96b), and lock information (97a, 97b). Therefore, when a data mover services a network client request for access to a locked file, there is no cached disk array overhead in managing the lock, because all of the required locking information is already in the local directory of the data mover. For fast access, the local directory (94a, 94b) of locking information from locked network files is kept in semiconductor buffer cache memory (62 in FIG. 5) of the data movers.

Detailed Description Text (55):
If the requested file access is not precluded by a lock, then in step 116 a message is broadcast over the Ethernet link (26) to the other data movers providing access to the file. These other data movers record the lock in their local directories. If the requested file access is found in step 116 to be precluded by a lock, then in step 118 a lock denied message is broadcast to the other data movers providing access to the file. In step 119, each of the data movers providing access to the file places the lock denied message on a local wait list for the file. Next, in step 120, a lock denied status message can be returned to the network client having requested file access, to indicate that there will be a delay in providing file access due to conflicting locks. Then, in step 121, the file access task is suspended until the lock is granted.

Detailed Description Text (60):
The network file manager program 141 is a conventional network file manager program that is modified for use with the data mover file manager program 142. For example, a suitable conventional network file manager program is available from Sun Microsystems Inc. The conventional network file manager program recognizes the file to logical block mapping 95a for reading and writing to the logical blocks. The conventional network file manager program also recognizes the file attributes 96a and manages network client ownership of file locks. The conventional file manager program, however, has no knowledge of the different data movers in the network file server, since the conventional file manager program is written for a server in which the conventional file manager program services all network file access requests recognized by the server.

Detailed Description Text (61):

In addition to the client ownership of file locks 143, the network file server including the data mover 21a has data mover ownership of file locks 144. In addition, the amount of locking information exchanged between the data movers over the Ethernet (26 in FIG. 2) can be reduced considerably by replicating in the data movers only the data mover ownership of file lock information and not the client ownership of file lock information. Therefore, if a network client were to open a file for a write operation by accessing the file from one data mover, the client would not be able to simultaneously access the file from another data mover. In practice, this limitation is insignificant in comparison to the increase in performance obtained by not exchanging or replicating client ownership information. Another advantage is that by not replicating client ownership information, the data mover file manager program 142 can be relatively independent from the network file manager program 141. The network file manager 141 manages the client ownership of the file locks 143 substantially independent of the data mover ownership of the file locks 144, and the data mover file manger 142 manages the data mover ownership of file locks substantially independent of the client ownership of file locks. Moreover, the network file manager 141 is primarily responsible for communication with network clients directing requests to the data mover, and the data mover file manager 142 is primarily responsible for communicating with other data movers by exchanging messages over the Ethernet (26 in FIG. 2).

Detailed Description Text (64):
In step 154, the data mover file manager obtains data mover ownership of the file with broadcast and replication of the data mover file ownership in the local file directories of the other data movers permitting access to the file, corresponding to steps 116 and 122 of FIG. 9. If the file is locked, and if there is no prior request on the local wait list and the file lock is owned by the data mover, or if the immediately prior request on the local wait list is a request of the data mover, then there is no need to broadcast a "lock denied" request to other data movers to ensure fair servicing of waiting client requests on a first come, first serve basis. Otherwise, if the file is locked, then the data mover file manager broadcasts a "lock denied" request in order to place the request on the wait lists of the other data movers to ensure fair servicing of the request. The "lock denied" or "lock granted" messages are broadcast over the Ethernet among the data movers with identification of the data mover originating the request, and without any identification of the client originating the request, corresponding to steps 123-125 in FIG. 9. Once file access is finished, execution continues to step 156.

Detailed Description Text (68):
As described above with reference to FIG. 6, one of the file access protocols desirable for use in a network file server is NFS, and one of the physical file systems desirable for use in a network file server is the UNIX File System (UFS).

Detailed Description Text (69):
NFS Version 2 has synchronous writes. When a client wants to write, it sends a string of write requests to the server. Each write request specifies the client sending the data to be written, a file identifier, and an offset into the file specifying where to begin the writing of the data. For each write request, the server writes data and attributes to disk before returning to the client an acknowledgement of completion of the write request. (The attributes include the size of the file, the client owning the file, the time the file was last modified, and pointers to the locations on the disk where the new data resides.) This synchronous write operation is very slow, because the server has to wait for disk I/O before beginning the next write request.

Detailed Description Text (70):
NFS Version 3 has asynchronous writes. In the asynchronous write protocol, the client sends a string of write requests to the server. For each write request, the server does a "fast write" to random access memory, and returns to the client an acknowledgment of completion before writing attributes and data to the disk. At

some point, the client may send a commit request to the server. In response to the commit request, the server checks whether all of the preceding data and attributes are written to disk, and once all of the preceding data and attributes are written to disk, the server returns to the client an acknowledgement of completion. This asynchronous write protocol is much faster than the synchronous write protocol. However, there is a data security problem with its implementation in a UNIX server.

Detailed Description Text (71):
In any kind of conventional UNIX server 200, as illustrated in FIG. 13, data passes through a number of layers 201, 202, 203 from a client 204 to disk storage 205. These layers include a file system layer 201 which maps file names to data storage locations, a storage layer 202 which performs cache management such as setting write pending flags, and a buffer cache layer 203 where data is stored in random access semiconductor memory.

Detailed Description Text (73):
If the new data is written to disk storage 205 before the new attributes and the server crashes, then upon recovery, everything in the buffer cache 203 may be lost. An attempt is therefore made to recover from whatever can be found on disk 205. The attributes are found and decoded to obtain pointers to data. The file may be corrupted if not all of the new attributes were written to disk. Some old attributes on the disk may point to old data, and some new attributes on the disk may point to new data.

Detailed Description Text (74):
If the new attributes are written before the new data and the server crashes, then upon recovery, the new attributes are found and decoded to obtain pointers to data. The file may be corrupted if not all of the new data were written to disk. In addition, the pointers for the new data not yet written may point to blocks of data from an old version of a different file. Therefore, the data security problem may occur, since the client owning the file being accessed may not have access privileges to the old version of the different file.

Detailed Description Text (75):
The asynchronous write security problem is solved by a modified server implementing a file system cache protocol. As shown in FIG. 14, a modified server 210 also passes data from a client 214 to disk 215 through a file system layer 211, a storage layer 212, and a buffer cache 213. In addition, the modified UNIX server 210 has file system cache 216. Data 217 and attributes 218 are stored in the file system cache of each data mover and are not written down to storage until receipt of a commit request from the client 214. When the commit request is received, the data 217 is sent before the attributes 218 from the file system cache to the storage layer 212.

Detailed Description Text (76):
The modified server 210 is constructed so that the order in which the file data 217 and the file attributes 218 are written from the file system cache 216 to the storage layer 212 is the order in which the file data 219 and file attributes 220 are written to nonvolatile storage. In other words, if file attributes are found in storage upon recovery, then so will the corresponding file data. This can be done in a number of ways. For example, all of the data and attributes written to the storage layer 212 are written to the buffer cache 213, and then the file data 219 in the buffer cache 213 are written to the disk 215 before the file attributes 220 are written to the disk 215. Upon recovery, the file data 221 and the file attributes 222 are read from the disk 215. Alternatively, the buffer cache 213 can be nonvolatile, battery-backed semiconductor memory, so that the order in which attributes and data are written from the buffer cache 213 to the disk 215 does not matter.

Detailed Description Text (77):
A flowchart of the operation of the modified server for servicing a read-write file access from a client is shown in FIG. 15. This flowchart represents control logic in the file system layer. In a first step 241, the file system layer of the server receives the client request and accesses a file directory in the file system layer and obtains a write lock to open the file. Next, in step 242, the file system layer writes new file data from the client and new file attributes to the file system cache, but does not write the new file data and new file attributes down to the storage layer. The file system may continue to write new file data and new file attributes to the file system cache until a commit request 243 is received from the client. When a commit request is received, as tested in step 243, then in step 244, the new file data written into the file system cache in step 242 is written from the file system cache to storage. Thereafter, in step 245, the new file attributes written into the file system cache in step 242 are written from the file system cache to storage. Thereafter, in step 246, the file system sends to the client an acknowledgement of completion of the commit operation.

Detailed Description Text (79):
The file system level cache protocol of FIG. 15 is best implemented in the network server 20 of FIG. 2 by incorporating the file system level cache (216 of FIG. 14) in the buffer cache (62 in FIG. 5) of semiconductor random access memory of each of the data movers 21 of FIG. 2. In this case, the new file attributes and the new file data are indexed by the file directory 94a in FIG. 11. The protocol of FIG. 15 is programmed into a UFS physical file system 79 of FIGS. 5 and 6. The storage layer 212, buffer cache 213 and disk 215 of FIG. 14 are in the cached disk array 23 of FIG. 2. In particular, the storage layer 212 is comprised of the channel directors 43 in FIG. 3, the buffer cache is comprised of the cache memory 41 of FIG. 3, and the disk 215 is comprised of the disk array 47 of FIG. 3.

Detailed Description Text (85):
VI. Message Collector Queue For Connection Oriented Protocols

Detailed Description Text (86):
As described above, the network file server 20 of FIGS. 1 and 2 supports a number of file access protocols 75 in FIG. 5. These file access protocols use a number of communication protocols, including the User Datagram Protocol (UDP) and the Transmission Control Protocol (TCP).

Detailed Description Text (87):
As illustrated in FIG. 16, UDP is a connectionless protocol. There is one fast pipe 261 conveying messages 262 (e.g., requests) from a number of clients 263 to a server 264. As used herein, the term "pipe" denotes generally a network link or message stream received by the file sever from one or more network clients. The messages 262 (represented individually by respective circle, triangle, and square icons) get mixed together during transmission in the pipe 261, and at the server 264 the messages are serviced by respective code threads 265.

Detailed Description Text (88):
As illustrated in FIG. 17, TCP is a connection oriented protocol. Each of the clients 273 is assigned a separate pipe 271 for sending messages 272 to the server 274, and each pipe 271 is serviced by a respective code thread 275.

Detailed Description Text (89):
In the UDP case, code threads are assigned to respective messages, and there are lots of code threads to service the client messages.

Detailed Description Text (90):
In the TCP case, the threads are assigned to respective pipes, and the pipes are assigned to respective clients. Therefore, the threads remain connected to the respective clients in the case of a series of messages from each client. There are

fewer TCP threads, and some threads are very busy and others are not very busy, since the threads remain connected to the clients. As a result, there is less balance; some threads work harder than others, and there is a loss of performance. This occurs in conventional NFS servers, such as NFS servers from Sun Microsystems Inc. and Digital Equipment Corp.

Detailed Description Text (91):
In order to minimize the loss of performance due to thread imbalance, a collector queue is used in a file server in order to combine messages from UDP and TCP streams. As shown in FIG. 18, threads 285 of a server 284 receive messages directly from the collector queue 286 rather than individual pipes or streams 281 conveying messages 282 from the clients 283. The messages 282 are received from the pipes or streams 282 by network link drivers 287 and placed in the collector queue 286. In the collector queue 286, messages from a pipe for a connection oriented process such as TCP are mixed and interleaved with messages for other connection oriented and connectionless processes. However, the collector queue 286 maintains the ordering of the messages in each pipe. For example, the collector queue 286 is serviced on a first-in, first-out basis. Any idle thread 289 can pick up a message from the collector queue and become an active thread 290 responding to the message. The threads 285 are components of the software implementing file access protocols 288. The collector queue 286 keeps track of which pipe 281 each message came from, and the reply of the server to each message is directed to the same pipe from which the message came from. Therefore, the collector queue 286 ensures balance and efficiency.

Detailed Description Text (95):
Each link pointer points to the link pointer of a next entry in the list 301 if there is a next entry, and otherwise the link first step 311 of FIG. 20, the code thread checks whether or not the collector queue is empty. If so, then in step 312 execution of the code thread is suspended (i.e., the thread becomes inactive) for a certain time, and later execution resumes by looping back to step 311. If in step 311 the collector queue is not empty, then execution continues to step 313. In step 313, the entry at the head of the collector queue is removed from the queue to obtain a message pointer and a corresponding pipe pointer. In step 314, the message pointer is used to obtain the corresponding message from the message buffer. In step 315, the message is interpreted and an appropriate reply is prepared. In step 316, the reply is sent to the pipe indicated by the pipe pointer, for transmission to the client that originated the message. Then in step 317 the memory of the collector queue entry removed in step 313 is deallocated and the memory of the message in the message buffer is deallocated, for example, by placing pointers to the collector queue entry and the message words onto free memory lists.

Detailed Description Text (99):
Turning now to FIG. 20, there is shown a flowchart of the programming for one of the code threads (285 in FIG. 18). In a continues to step 323. In step 323, the link driver allocates memory in the message buffer to store the message. Next, in step 324, the link driver puts the message in the message buffer. Then in step 325 the link driver allocates memory for the collector queue entry (in the list 301 of FIG. 19), and in step 326 the link driver puts the pointers (i.e., the message pointer, pipe pointer, and link pointer) into the collector queue entry.

Detailed Description Text (100):
Now that the message has been inserted into the collector queue, in step 327 the link driver checks whether there is an idle thread. If not, then the link driver is finished processing the message, and the message in the queue will be picked up and serviced by one of the threads that is currently busy servicing another message. If in step 327 the link driver finds an idle thread, then in step 328 the link driver activates execution of the idle thread, and the link driver is finished processing the message. In this case, the activated idle thread will service the message that was just inserted into the collector queue.

Detailed Description Text (101):
Steps 312, 327 and 328 can be implemented using a callback mechanism. A callback function is registered by a pipe when the pipe is opened. When an idle server thread tries to pick up a reference to a pipe from the head of the collector queue and the queue is empty, the thread is blocked on a condition variable and can only be resumed when a signal is sent by the callback function to the condition variable. The callback function is invoked as soon as the pipe detects an incoming message. Invocation of the callback function sends a signal to the condition variable, causing resumption of one of any number of threads blocked on the condition variable.

Detailed Description Text (102):
In the network server 20 of FIG. 2, a respective collector queue is used in each data mover to queue all client messages received by the data mover. The collector queue is part of the communication stacks 74 in FIG. 5 residing between the network link drivers 72 assigned to the pipes or network links, and the code threads which are part of the file access protocols 75. The idle code threads are activated by the real-time scheduler in the kernel 63 in FIG. 5.

Detailed Description Text (103):
In view of the above, there has been described a network file server having a collector queue that minimizes loss of performance due to thread imbalance when servicing messages from a connection oriented process concurrent with other processes. The collector queue combines messages from the connection oriented process with messages from the other concurrent processes. The threads of the server receive messages from the collector queue rather than individual pipes. Any idle thread can pick up a message from the collector queue. Therefore the collector queue ensures thread balance and efficiency in servicing the messages.

Other Reference Publication (9):
Storage Computer Corporation, "High Performance, Fault Tolerant Disk Array Platform for File Servers and Computer Systems," 1991, Nashua, NH, 12 pages.

Other Reference Publication (10):
Krishnan Natarajan, "Video Servers Take Root," IEEE Spectrum, Apr. 1995, IEEE, New York, NY, pp. 66-69.

Other Reference Publication (16):
"EMC Moves Into Servers," Broadcasting Cable, Apr. 15, 1996.

Other Reference Publication (21):
Vin HM, Rangan PV, (1993), "Designing a multiuser HDTV storage server," IEEE Journal on Selected Areas in Communication, vol. 11, No. 1, Jan. 1993, pp. 153-164.

Other Reference Publication (24):
Haskin R, The shark continuous-media file server, Proceedings IEEE COMPCON 93, San Francisco, Calif., 1993, pp. 12-15.

Other Reference Publication (26):
Lougher, P, Sheperd, D. "The design of a storage server for continuous media," The Computer Journal, vol. 36, No. 1, 1993, pp. 32-42.

Other Reference Publication (29):
Tobagi FA, Pang J, "StarWorks (Trademark)--A video applications server," Proceedings, IEEE COMPCON 93, San Francisco, Calif., 1993, pp. 4-11.

CLAIMS:

h e b b g e e e f c e e ge

1. A method of operating a file server to service messages of concurrent connection oriented processes and connectionless processes, the messages originating from a plurality of clients in a data network, the messages being sent in a plurality of pipes to the file server, including pipes conveying messages of the connection oriented processes and at least one pipe conveying messages of the connectionless processes, said method comprising the steps of:

(a) the file server receiving messages from the pipes and placing them in a queue combining messages of the connection oriented processes with messages of the connectionless processes, and placing in the queue with each message an indication of the pipe from which the file server received said each message; and

(b) the file server receiving the messages from the queue and servicing the messages by interpreting the messages, formulating replies, and directing the replies to the pipes from which the file server received the messages for transmission to the clients from which the messages originated.

2. The method as claimed in claim 1, wherein the connectionless processes are User Datagram Protocol (UDP) processes, and the connection oriented processes are Transmission Control Protocol (TCP) processes.

3. The method as claimed in claim 1, wherein the file server places the messages in the queue in order as the file server receives the messages from the pipes, and the file server removes the messages from the queue in a first-in, first-out order for servicing.

4. The method as claimed in claim 1, wherein the queue includes a list and a message buffer, and wherein the file server places each message in the queue by placing said each message in the message buffer and placing a pointer on the list to said each message in the message buffer.

5. A method of operating a file server to service messages of a connection oriented process concurrent with other processes, the messages originating from a plurality of clients in a data network, the messages being sent in a plurality of pipes to the file server, including a pipe conveying messages of the connection oriented process, said method comprising the steps of:

(a) the file server receiving messages from the pipes and placing them in a queue combining messages of the connection oriented process with messages of the other processes, and placing in the queue with each message an indication of the pipe from which the file server received said each message; and

(b) threads of the file server receiving the messages from the queue, each of the threads servicing a respective message from the queue by interpreting the message, formulating a reply, and directing the reply to the pipe from which the file server received the respective message.

6. The method as claimed in claim 5, wherein the file server places the messages in the queue in order as the file server receives the messages from the pipes, and the messages are removed from the queue in a first-in, first-out order for servicing by the threads.

7. The method as claimed in claim 6, wherein each thread finished with servicing a message checks whether the queue is empty, and when said each thread finds that the queue is not empty, the thread services a next message from the queue, and when said each thread finds that the queue is empty, the file server suspends execution of the thread.

8. A method of operating a file server to service messages of connection oriented processes and connectionless processes concurrent with the connection oriented

processes, the messages originating from a plurality of clients in a data network, the messages being sent in a plurality of pipes to the file server, including respective pipes conveying messages of the connection oriented processes and at least one pipe conveying messages of the connectionless processes, said method comprising the steps of:

(a) the file server receiving messages from the pipes and placing them in a queue combining messages of the connection oriented processes with messages of the connectionless processes, and placing in the queue with each message an indication of the pipe from which the file server received said each message; and

(b) threads of the file server receiving the messages from the queue, each of the threads servicing a respective message from the queue by interpreting the message, formulating a reply, and directing the reply to the pipe from which the file server received the respective message.

9. A file server servicing clients in a network, said file server comprising, in combination:

a) first means for receiving messages transmitted by one client in a connection oriented process;

b) second means for receiving messages transmitted by other clients in other processes concurrent with the connection oriented process;

c) a queue for storing messages received by the first means for receiving and the second means for receiving; the first means for receiving being connected to the queue for placing in the queue the messages received from said one client when the messages received from said one client are received, and the second means for receiving being connected to the queue for placing in the queue the messages received from said other clients when the messages received from said other clients are received from said other clients; and

d) means for servicing the messages in the queue.

10. The file server as claimed in claim 9, wherein the first means for receiving includes a first network link driver for a network link connecting said one client to the file server for communication in the connection oriented process, and the second means for receiving includes a second network link driver for a network link connecting a plurality of clients to the file server for communication in connectionless processes.

11. The file server as claimed in claim 10, wherein the connection oriented process is a Transmission Control Protocol (TCP) process, and the connectionless processes are User Datagram Protocol (UDP) processes.

12. The file server as claimed in claim 9, wherein each of the means for receiving includes means for storing, in the queue, in association with each message, an indication of the means for receiving that received said each message, and wherein the means for servicing includes means for reading the indication associated with said each message for directing a reply to the means for receiving that received said each message.

13. The file server as claimed in claim 9, wherein the means for servicing includes a plurality of threads, each message in the queue being serviced by a respective one of the threads.

14. The file server as claimed in claim 13, wherein the means for servicing includes means for activating an idle one of the threads for servicing a message placed in the queue.

15. The file server as claimed in claim 14, wherein the queue includes a message buffer storing the messages in the queue, and a list of pointers to the messages stored in the message buffer.

16. A file server for servicing messages of a connection oriented process concurrent with other processes, the messages originating from a plurality of clients in a data network, the messages being sent in a plurality of pipes to the file server, including a pipe conveying messages of the connection oriented process, said file server comprising, in combination:

(a) a queue for storing messages of the connection oriented process combined with messages of the other processes, and for storing an indication of the pipe from which the file server received each message; and

(b) a plurality of threads, each of the threads comprising a program executable in the file server for servicing a respective message from the queue by interpreting the message, formulating a reply, and directing the reply to the pipe from which the file server received the respective message.

17. The file server as claimed in claim 16, wherein each thread further includes programming executable after servicing a message for checking whether the queue is empty, and when the queue is found to be not empty, servicing a next message from the queue, and when the queue is found to be empty, suspending execution of the thread.

18. The file server as claimed in claim 17, wherein the file server includes means for activating a suspended one of the threads for servicing a message placed in the queue.

19. The file server as claimed in claim 16, wherein the connection oriented process is a Transmission Control Protocol (TCP) process, and the connectionless processes are User Datagram Protocol (UDP) processes.

20. The file server as claimed in claim 16, wherein the queue includes a message buffer and a list of pointers to the messages stored in the message buffer.

21. The file server as claimed in claim 20, wherein the list is a singly-linked list.

22. A method of operating a file server to service messages of a connection oriented process concurrent with other processes, the messages originating from a plurality of clients in a data network, the messages being sent in a plurality of pipes to the file server, including a pipe conveying messages of the connection oriented process, said method comprising the steps of:

(a) the file server receiving messages from the pipes and placing them in a queue combining messages of the connection oriented process with messages of the other processes; and

(b) the file server obtaining messages from the queue and servicing the messages obtained from the queue.

23. The method as claimed in claim 22, wherein the file server places in the queue, in association with the messages, indications of the pipes from which the messages are received, and wherein the file server reads from the queue the indications of the pipes from which the messages are received in order to direct replies to the messages to the pipes from which the messages are received.

24. The method as claimed in claim 22, wherein the file server combines in the

queue messages of connectionless processes with messages of connection oriented processes.

25. The method as claimed in claim 24, wherein the connectionless processes are User Datagram Protocol (UDP) processes, and the connection oriented processes are Transmission Control Protocol (TCP) processes.

26. The method as claimed in claim 22, wherein the file server places the messages in the queue in order as the file server receives the messages from clients in a data network, and the file server obtains the messages from the queue in a first-in, first-out order for servicing.

27. The method as claimed in claim 22, wherein the queue includes a list and a message buffer, and wherein the file server places each message in the queue by placing said each message in the message buffer and placing, in the list, a pointer to said each message in the message buffer.

28. A method of operating a file server to service messages of a connection oriented process concurrent with other processes, the messages originating from a plurality of clients in a data network, the messages being sent in a plurality of pipes to the file server, including a pipe conveying messages of the connection oriented process, said method comprising the steps of:

(a) the file server receiving messages from the pipes and placing them in a queue combining messages of the connection oriented process with messages of the other processes; and

(b) threads of the file server receiving the messages from the queue, each of the threads servicing a respective message from the queue.

29. The method as claimed in claim 28, wherein the file server places in the queue, in association with each message, an indication of the pipe from which said each message is received.

30. The method as claimed in claim 28, wherein each of the threads services the respective message from the queue by interpreting the message and formulating a reply.

31. The method as claimed in claim 28, wherein the file server combines in the queue messages of connectionless processes with messages of connection oriented processes.

32. The method as claimed in claim 28, wherein the file server places the messages in the queue in order as the file server receives the messages from clients in a data network, and the messages are removed from the queue in a first-in, first-out order for servicing by the threads.

33. The method as claimed in claim 28, wherein each thread finished with servicing a message, checks whether the queue is empty, and when said each thread finds that the queue is not empty, the thread services a next message from the queue, and when said each thread finds that the queue is empty, the file server suspends execution of the thread.

34. The method as claimed in claim 28, wherein the queue includes a list and a message buffer, and wherein the file server places each message in the queue by placing said each message in the message buffer and placing, in the list, a pointer to said each message in the message buffer.

First Hit Fwd Refs
**End of Result Set**

Generate Collection Print

L19: Entry 9 of 9 File: USPT Sep 8, 1998

DOCUMENT-IDENTIFIER: US 5805804 A
TITLE: Method and apparatus for scalable, high bandwidth storage retrieval and transportation of multimedia data on a network

Abstract Text (1):
An improved system and method for providing multimedia data in a networked system is disclosed. The present invention provides a platform for distributed client-server computing and access to data over asymmetric real-time networks. A service mechanism allows applications to be split such that client devices (set-top boxes, personal digital assistants, etc.) can focus on presentation, while backend services running in a distributed server complex, provide access to data via messaging across an abstracted interface.

Brief Summary Text (9):
Currently, most data accessed on large servers is structured data stored in traditional databases. Networks are local area network (LAN) based and clients range from simple terminals to powerful workstations. The user is corporate and the application developer is an MIS professional.

Brief Summary Text (13):
The last decade of computing has produced inexpensive client hardware with shrink-wrapped software, scalable server hardware with complex data management software, and ubiquitous heterogeneous networking hardware with sophisticated networking software. However, the promise that multimedia data will be readily shared and easy to access has been mostly unfulfilled. Most conventional multimedia software is single user and fairly easy to use or it allows resources to be shared but the degree of sharing extracts a correspondingly high price in usability. Consumer-based interactive networking is an attempt to provide simple access to unprecedented amounts of shared data. The present invention provides a framework for such an endeavor.

Brief Summary Text (16):
The present invention is a better means and method for providing multimedia data in a networked system. The present invention provides a platform for distributed client-server computing and access to data over asymmetric real-time networks. A service mechanism allows applications to be split such that client devices (set-top boxes, personal digital assistants, etc.) can focus on presentation, while backend services running in a distributed server complex, provide access to data via messaging across an abstracted interface. Services enable clients to access data or resources that the clients cannot (or should not) access directly. Each service provides access to a particular type of data or resources. A service exports one or more functions, which perform specific actions related to the data or resource. A client program invokes a function by communicating with the service that exports that function.

Brief Summary Text (17):
The present invention supports access to all types of conventional data stored in conventional relational and text databases. In addition, the present invention

includes a real-time stream server that supports storage and playback of real-time audio and video data. The Media Server of the present invention also provides access to data stored in file systems or as binary large objects (BLOBs-images, executables, etc.).

Drawing Description Text (3):
FIG. 2 illustrates the internal architecture of the media server of the present invention.

Drawing Description Text (5):
FIG. 4 illustrates the diverse network pathway that may exist between clients and servers.

Drawing Description Text (7):
FIG. 6 illustrates the upstream manager interfaces in the media server.

Drawing Description Text (8):
FIG. 7 illustrates the content of the Connection Service Table.

Drawing Description Text (10):
FIGS. 9-10 are flowcharts illustrating the processing flow for establising a connection between a client and the media server.

Drawing Description Text (11):
FIGS. 11-12 are flowcharts illustrating the processing flow for accessing a service thorough the media server.

Detailed Description Text (3):
The basic architectural components of the consumer-based interactive network architecture of the present invention is illustrated in FIG. 1. The media server 100 consists of any number of computers networked in any fashion. The network 120 connecting clients 110 and servers 100 are asymmetric, with high bandwidth available in the downstream direction 124. Client devices 110 are conventional and well known systems generally built for interactive TV. These client devices are generically denoted set-top boxes. Other classes of devices (personal digital assistants, video phones, etc.) will become important to consumer-based networks in the near future. The present invention supports these devices as well.

Detailed Description Text (5):
The major characteristic shared by the networks currently being deployed for interactive TV is asymmetric bandwidth. Downstream bandwidth (i.e. the link from server to client device) ranges from a minimum of 1.5 megabits/sec (DS1 data rates) to 45 megabits/sec (DS3 data rates). Upstream bandwidth on a back channel (i.e. the link from client device to server) may be more modest, ranging from 9600 bits/sec to 64 kilobits/sec. Bandwidth will increase in both directions, but will probably remain highly asymmetric; because, most information flows toward the consumer.

Detailed Description Text (9):
ATM (Asynchronous Transfer Mode) to the home over fiber/coax hybrid networks provides maximum flexibility for both bandwidth and addressing. ATM technology is well known in the art. Because ATM provides dynamic allocation of bandwidth, it is possible to provide variable bandwidth bit streams using ATM. Thus, it is possible to serve movies at 3 megabits/sec, sporting events encoded in real-time at 8 megabits/sec, and HDTV compressed at 20 megabits/sec (or whatever bandwidth is required). ATM also provides flexibility at the headend since many servers can be directly connected into the ATM switch fabric at very high bandwidth; 155 megabit/sec is available today, increasing to 622 megabits/sec or 1.2 gigabits/sec over time.

Detailed Description Text (22):

Thus, integrating video compression into the heart of the Media Server is essential. The present invention accomplishes this by having the Stream Service 224 generate and manipulate the real-time audio/video data as it served to the downstream manager. This includes identification and modification of the data structures (headers, pictures, aspect ratios, etc.) in the original data in real-time.

Detailed Description Text (25):
The network interface provides both downstream and upstream interfaces over one or more physical connections.

Detailed Description Text (37):
In view of these prior art subsystems and the need to support an additional level of functionality, the media server 100 and the supporting network layer of the present invention provides a full-featured system for interactive multimedia presentations.

Detailed Description Text (38):
The Architecture of the Media Server of the Present Invention

Detailed Description Text (39):
The present invention provides a layer of software that enables distributed client-server computing in the consumer-based networks described above. The main components of the present invention include:

Detailed Description Text (43):
a real-time stream server

Detailed Description Text (45):
Referring now to FIG. 1 and FIG. 2, the architecture of the present invention assumes that data is stored and managed on the server 100 and the client device 110 provides a view onto that data. Generally, the view is through some paradigm such as a digital mail, personal digital assistant, or electronic newspaper. The user navigates locally on the client device 110 and data is requested from the server 100 as necessary. This provides a very clean split between the client side of the application and the server side.

Detailed Description Text (46):
Server applications (services) 122 are "data based" and developed with the same tools used to build corporate databases (i.e. data modeling tools, schema editors, etc.). These services must be built in a reliable and scalable manner.

Detailed Description Text (47):
Client applications 276, shown in FIG. 2, are built using interactive, graphical authoring tools that allow digital assets (video, images, sounds, etc.) to be mixed with presentation logic to produce a runtime environment for interactive TV. The present invention supports runtime environments by providing such services as application download, asset management, authorization, and stream interaction (as will be described below). Typically, there are many more client applications built than those for the server 100. For instance, there may be three home shopping services nationwide that are used by hundreds of client applications.

Detailed Description Text (48):
Because latency is a major issue in the network 120 as described earlier, the architecture of the present invention allows distributed applications to be built where all state information is maintained on the client device 110. The use of remote procedure calls (RPC) to access services and data through the media server 100 is preferred over the use of a traditional Structured Query Language (SQL) for data access. This is because it reduces the number of round-trip messages and provides easy to use interfaces to application services.

Detailed Description Text (50):
FIG. 1 shows a plurality of available services 122 on media server 100. FIG. 2 shows examples of these services and shows in more detail how these services can be partitioned into applications services 240 and system services 214. However, in reality, this partitioning is purely arbitrary. Any of the services 240 or 214 are accessible to a client on client device 110 or accessible to another service on server 100.

Detailed Description Text (51):
Each service 122 as shown in FIG. 1 comprises one or more cooperating server execution threads which may be distributed across several machines. Load balancing across these server threads is performed by server 100 dynamically and transparently to the client. Requests sent by a client are routed to a service control point which decides, based on current system activity, which server process can actually handle the request. The service control points support many server threads before they become bottlenecks. Still, because these systems must be highly scalable, service control points may be replicated. For example, the name service 222, which is used to locate all other servers, must be replicated in order to handle the large numbers of requests sent to it.

Detailed Description Text (52):
To shield client applications from network implementation details, client applications never interact directly with the underlying databases. Instead, all applications developed with the media server 100 communicate with services by sending messages locally or remotely.

Detailed Description Text (53):
Interfaces to the services are defined using an interface definition language (IDL). A service interface consists of a set of operations that define what the service can do. Once created, the interface definition is compiled to generate stubs which isolate the distributed nature of the system from the computations being performed. Client applications 276 execute the operations by making remote procedure calls (RPC) to the server 100. The use of RPC in the present invention is described in more detail below. In addition to providing its base functionality to client applications 276, each service 214 and 240 has a standard interface for configuration, management, monitoring, debugging, logging and auditing.

Detailed Description Text (59):
Structured data is stored in the Text database 253 and accessed via SQL and PL/SQL (Procedural Language/SQL). Text database 253 provides distribution, replication, and parallel access to the data. Stored procedures, executing within the protected space of the database environment, provide an excellent mechanism for building reliable services. A server application in this model consists of a schema and a set of procedures to access the data in the schema (which may be invoked directly from the client via the procedural service described below).

Detailed Description Text (60):
Real-time Stream Server

Detailed Description Text (62):
Therefore, the real-time components of the media server 100 are segmented from the other parts by a scheduling "fire-wall". All access into the real-time section of the server goes through a real-time scheduler 299 which analyzes the load any given request will make on the system, determine if the request can be granted given the current system load, and then schedule the access. The real-time scheduler takes CPU, disk and memory resources into account when analyzing a request.

Detailed Description Text (63):
Even with the scheduler ensuring that the system does not become over-committed,

there are many constraints on the real-time server design and operation. The server must:

Detailed Description Text (66):
store an enormous amount of data and coordinate movement of that data between different media and different servers

Detailed Description Text (68):
be portable to any viable server hardware platform

Detailed Description Text (70):
The real-time stream server provides full VCR-like controls to the user: fast forward and rewind, slow forward and rewind, frame advance and rewind, random positioning, etc. However, it is not simple to fit these features into a real-time scheduling system, since each places a different load upon hardware resources. In addition, depending on the video/audio codec being used, each of these special modes may place different demands upon the real-time scheduler.

Detailed Description Text (71):
In the simplest model, a stream is merely a string of contiguous bits which must appear at the decoder at a particular rate. The server 100 is responsible for accepting a command to start a stream and one to prematurely terminate a stream. In this model, streams would be easy to allocate, schedule and deliver. Maximum service levels could be easily computed.

Detailed Description Text (72):
Unfortunately, the real world is not so simple. During the playback of a stream, many events may require the stream server to change its behavior. For example, if the user presses the pause button on the remote control, the server 100 must stop delivering bits temporarily. Since there are network buffers between the server and the set-top box, the server 100 has no way of determining exactly where the set-top box 110 was paused. So the server 100 and set-top box 110 must communicate so that the server 100 may queue from that point so that it will be ready for the subsequent play command. The present invention provides this position coordination through a real-time stream control interface which resides both on the client and server.

Detailed Description Text (75):
By providing random entry ability and controlled playback of data, the stream service not only provides rich real-time stream playback but also is part of the hypermedia interactive environment provided by a client application. Thus, in addition to playing movies at a user's request, the server 100 can play video or audio clips in response to other services, such as home shopping, education and video hyperlinks.

Detailed Description Text (78):
As information servers take on more and more of the communication tasks of consumers, they will have to become as reliable as the current telephone system. To achieve this, real-time server 100 can be configured for various levels of reliability.

Detailed Description Text (79):
By far the most common failure point in the isochronous server 100 is a magnetic disk drive. This is largely because the disk drives are the only part of the core system with moving parts. Even with mean time between failures (MTBF) of 1,000,000 hours, systems with large farms of disks will experience individual disk failure on a regular basis. For a system with many terabytes on-line, and thousands of disk drives, frequent disk drive failures could render large amounts of content inaccessible.

Detailed Description Text (80):
The real-time server 100 can identify and correct disk failures without interrupting the real-time data flow. The approach used is described in an above referenced co-pending patent application entitled "Real-Time RAID." A new approach is necessary; because, traditional approaches to disk redundancy (e.g. RAID) are anything but real-time. Disks also go off-line many times during the day due to thermal recalibration, predictive failure analysis checks, etc. As with disk failures, the present invention rides through these temporary interruptions without any interruption of real-time service. On any system with disks that may be hot-swapped, when the media server 100 has determined that a disk has failed and needs to be replaced, media server 100 prompts the operator to remove the bad disk canister and load in a new one. Media server 100 then requests resource bandwidth from the server scheduler to rebuild the volume as fast as possible without disturbing stream playback.

Detailed Description Text (81):
The present invention provides two dimensions of reliability configuration for disks. The system manager can select what storage overhead to incur to guard against both multiple simultaneous disk failures or multiple disk chain controller failures. The ability to continue uninterrupted play through both disk and controller failures solves the largest reliability challenge present in the real-time server 100.

Detailed Description Text (82):
The second most common failure is a network backbone communications failure, most likely from a piece of hardware outside the real-time server 100. To cope with this failure potential, the present invention can accept routing commands during stream playback. When the network management service 227 determines that a stream needs to be rerouted, network management service 227 commands the virtual network layer 212 to re-route the signal and simultaneously informs the stream server 224 where it should send the data. This happens on the fly, without having to restart the stream.

Detailed Description Text (83):
Software failure is a third potential source of system failure in the media server 100 of the present invention. For the most part, portions of the media server 100 are logically placed behind a real-time "firewall." These portions are thus independent from each other, so a single softw failure should interrupt at most a handful of streams. Because the realtime stream service 224 checkpoints frequently, a stream can be restarted from a different point in the system quickly with only a small service disruption to the user. Once the service manager detects the software fault, it immediately restarts the parts of the system that failed.

Detailed Description Text (84):
Hardware and software redundancy and independent restart-ability are part of the basic architecture of the media server 100. By placing these elements into every service, a complex system can be diagnosed and restored to normal operation in the shortest possible time.

Detailed Description Text (86):
Financially, the key metric in serving video is megabytes per second per dollar (more generally, bandwidth per time per cost). Because different content (feature-length movies, classic movies, home shopping videos) will have different revenue and usage patterns, the server 100 allows the data to be stored on a variety of devices, each with different cost, bandwidth, and capacity characteristics.

Detailed Description Text (87):
The system manages a multi-tiered media datastore 251 where movies might be staged from off-line tape to an on-line optical jukebox to magnetic disk and finally to RAM. With commodity disk drive prices currently at approximately $600/gigabyte, a

2-hour movie costs $800 to store on magnetic storage. This same movie costs $40,000 to store in RAM and about $50 to store on off-line tape. Where the content resides at any given time is determined by the server 100; but, a human server manager can override that decision at any time. When a decision is made to move a portion of media content, this request is scheduled into the realtime scheduler 299 just like user requests.

Detailed Description Text (88):
In addition to staging data between multiple tiers of storage on a single server, the media server 100 can also move between servers over a high-speed inter-server network backplane. Because of the design of the storage system, staging from another server is equivalent to retrieving from off-line tape. When data is moved between different storage media or between servers, the reliable messaging features of the network protocol of the present invention are used.

Detailed Description Text (89):
When media content is loading from a tape, the user is given control as soon as possible without waiting for the entire movie to load. While the movie is playing, media content is cached in either disk or RAM to provide full rewind and still capabilities. Of course, since the movie is loading from a sequential medium, fast forward is limited to the amount the server 100 has loaded ahead of the user's current frame.

Detailed Description Text (91):
Interconnect bandwidth is the limiting factor not only in the networks being deployed but also in the real-time server 100. Even with the high data and video bandwidths being used today, CPU power is plentiful. This bandwidth is divided into three categories: reading from external I/O devices such as disk or tape, internal routing of the data within the server 100, and the external routing of the data over the network. By balancing these three categories, the cost per stream can be minimized.

Detailed Description Text (92):
Building a well balanced system requires that it must be scheduled precisely, or one segment of it may temporarily become overloaded. As described above, the real-time scheduler 299 has a very complex job. It must mix bandwidth requests from streams in normal play-forward mode, others which are being fast-forwarded, disk rebuild processes, downstream application data, off-line media requests, and requests to and from other servers. The real-time scheduler 299 of server 100 allows the system to be used within a small margin of maximum capacity, yet still ensures that no stream will ever glitch.

Detailed Description Text (93):
It is transparent to the system administrator whether the system is serving thousands of streams of the same movie or thousands of streams of different movies: any combination is possible. The manager does not have to make decisions about disk layout in order to achieve better performance for one load pattern or another. In addition, if the network connected to the server 100 supports routing one stream to multiple households, the system can "piggyback" users onto other users already playing that stream at nearby points. Of course, this optimization becomes more complicated when one of the users hits the pause or fast-forward keys, because the server 100 must start a new stream to service the customer who split from the concurrent pack.

Detailed Description Text (95):
By requiring only a low-level real-time kernel, the media server 100 software is portable among many existing and new hardware platforms. The media server 100 can be delivered on a wide range of conventional systems, ranging from workstations serving a handful of streams through symmetric multiprocessor (SMP) machines up to massively parallel (MMP) machines delivering many thousands of streams. In

addition, since the present invention depends only on features that must be minimally present in all hardware, the media server 100 is operating-system independent.

Detailed Description Text (97):
The network protocol of the present invention provides the communication backbone that allows services scattered across heterogeneous, asymmetric networks to communicate with each other transparently. For the most part, the level of exposure of the network for application servers and clients is minimal, since RPC serves to hide these implementation specific details.

Detailed Description Text (113):
Referring now to FIG. 5, a basic asymmetric network configuration is illustrated. In order for a node 560 (Client 1 in this example) to obtain a logical network address for itself, the node 560 must know the data link 562 through which it can reach a server, and the link 566 through which it expects the response from the server. The latter downstream link 566 is the link that is actually assigned the address.

Detailed Description Text (114):
The address request from the node 560 specifies the downstream link 566 in a data-link specific manner. The problem is to figure out how to get to that data link assigned. The information provided by the node 560 may not be sufficient to enable completion of the address request. Often the server that receives the address request is not connected directly to the downstream link 566.

Detailed Description Text (115):
In FIG. 5, suppose that gateway/server 1 receives an address request for a client device 560 over a serial line 562. The client specified a T1 line as the downstream link. Unfortunately, the T1 line is not attached to the machine where gateway/server 1 is running; it happens to be attached to a different machine (gateway/server 2), which is connected to gateway/server 1 via Ethernet link 564. Gateway/server 1 forwards the address request over Ethernet link 564 to gateway/server 2, which allocates an address and issues the response to the requesting client device 560 through the downstream channel 566.

Detailed Description Text (116):
The network protocol of the present invention is able to forward address requests in this manner; because, gateways and servers know about each other and about the types of data links to which they are connected. This address requisition process is described in more detail below in connection with a description of the processing performed by the upstream manager 220 and downstream manager 210 of server 100.

Detailed Description Text (118):
Suppose the client 560 requested a second address, but this time specified the serial line 562 (now bidirectional) as the downstream link. This time, gateway/server 1 can assign an address itself; because, it is connected directly to the serial line 562. However, the client 560 now has a second address completely unrelated to the first address. This leads to an important point: only a server attached to the head of a link can legitimately assign an address for that link.

Detailed Description Text (119):
As a very useful side effect, the dynamic address allocation of the present invention causes the creation and maintenance of routing tables. Because the address request is self-identifying (even if only in data-link-specific terms), the server has enough information to enter new routes into the appropriate routing tables. Servers themselves may requisition entire subnets from larger-scale servers, thus serving to automate large amounts of network administration.

Detailed Description Text (123):
The transport layer must be able to adapt to these changes in order to maintain reliable and efficient transport. The conventional TCP/IP means of solving these problems are ineffective because they rely on the existence of a connection. Since most data traffic in these networks is connectionless (for reasons explained below), another solution must be used.

Detailed Description Text (125):
However, without connections, tracking round-trip times provides almost useless information. There is no reason to assume that the same round-trip will occur repeatedly. Nor does it help any of the other nodes in the network. Instead, the network protocol of the present invention focuses on trip times across individual links. This information is propagated through the network over time to allow individual nodes to have local access to trip estimate information. To avoid saturating the network with control messages, this information is usually piggy-backed inside data messages. As a benevolent side effect of the propagation, brief, spurious fluctuations in bandwidth have no noticeable impact on trip estimation.

Detailed Description Text (126):
The present invention provides an architecture in which this consumer model can evolve. By offering a service framework that supports server application development, the media server 100 allows developers to concentrate on the design of the media-based applications, such as interactive shopping, news, games, research and education. The network protocol of the present invention enables these applications to communicate transparently across the complex asymmetrical networks of today. Behind the services architecture, an isochronous server and traditional data access methods deliver an information-rich environment to the consumer via RPC.

Detailed Description Text (127):
Media Server Architecture

Detailed Description Text (128):
Referring again to FIG. 2, the architecture of the media server 100 is described in detail in the following sections. The media server 100 includes these components as shown in the FIG. 2:

Detailed Description Text (131):
connection service 230

Detailed Description Text (143):
The upstream manager 220 (USM) accepts messages from set-top boxes 110 and routes them to services on the media server 100. The upstream manager 220 also binds a downstream link for the messages in a manner described in more detail below.

Detailed Description Text (146):
The USM 220 and the DSM 210 are gateways that bridge two different types of networks. The USM 220 and DSM 210 move data packets (i.e., portions of messages) across the bridge from one type of network to another. These packets are routed independently from each other. The routing is based on the intended destination of the packets. These packets are portions of messages that may have originated from a client (set top box 110) or a server 100, such as in response to a remote procedure call RPC. In any case, in the preferred embodiment, neither the USM 220 nor the DSM 210 actually decodes the messages and processes them. The USM 220 and DSM do not attempt to assemble or disassemble these messages. The USM 220 and DSM 210 see each packet as an independent entity which is to be routed based on a destination address stored in the packet header.

Detailed Description Text (147):
Connection Service 230

Detailed Description Text (148):
The connection service 230 maintains a connection database that keeps track of all currently active circuits on media server 100 and their physical and logical addresses. When a set-top box 110 initially connects to the media server 100, the set top box 110 sends a message to the connection service 230 via the upstream manager 220. The connection service 230 then establishes a downstream manager 210 for the circuit and updates the connection service database.

Detailed Description Text (149):
The present invention is intended to be able to adapt itself to a variety of network topologies. Asymmetric networks complicate the process by which client programs connect to the media server 100. The Connection Service 230 is designed to encapsulate those complications.

Detailed Description Text (150):
The Connection Service 230 provides a generic control point for establishing connections between the media server 100 and its clients. It manages allocation and deallocation of asymmetric virtual circuits through the media server 100. The Connection Service 230 can be queried for information about its mappings and can inform interested parties of changes in connection state.

Detailed Description Text (151):
The Connection Service 230 is the single, reliable repository for address mappings. Other services may cache needed addresses for performance reasons; if they crash, though, those mappings can always be reconstructed by querying the Connection Service 230.

Detailed Description Text (152):
In an asymmetric network, upstream and downstream messages between a client and a server take place over distinct network interfaces. For example, messages from a client to a server may be routed over an X.25 packet-switched network, while responses from the server back to the client flow over a dedicated T1 line. This was described above in connection with FIG. 5.

Detailed Description Text (153):
Before a client can hold round-trip conversations using the media server 100, a virtual circuit between the upstream and downstream networks must be established. The Connection Service 230 is responsible for creating and managing these virtual circuits. Connection Service 230 makes the following assumptions about the nature of the network:

Detailed Description Text (155):
When a client wants to connect to the media server 100, a physical upstream channel--by which the client can address the server--will already exist.

Detailed Description Text (158):
It is up to deployment-specific conventional software in the virtual network layer 212 to manage physical upstream channels. It may be the case that some server process, under the direction of an external network control node, actually establishes contact with the client. Conceptually, however, this protocol is transparent to the media server 100 proper.

Detailed Description Text (162):
The logical address of a client. Because it is known that a single DSM 210 can only serve one client (and one virtual circuit) at a time, this logical address is pre-assigned when the DSM 210 is booted. The same logical address is recycled as clients connect to and disconnect from the media server 100.

Detailed Description Text (165):

The client's downstream physical address represents the far end of the connection for which a DSM physical address is the head end.

Detailed Description Text (168):
Connection establishment is highly deployment-specific, but consists of four distinct phases.

Detailed Description Text (169):
1. The client communicates with the upstream manager 220 of the media server 100.

Detailed Description Text (170):
2. The client communicates with the Connection Service 230 to establish a virtual circuit through a media server 100. This phase results in a round-trip path through the physical network to the client.

Detailed Description Text (172):
4. The client requests a logical client address for itself from the media server 100.

Detailed Description Text (174):
The Connection Service 230 can be queried by any other service that needs to map between resources or that wants to inquire about the state of a connection. For example, if the stream service 224 receives a network message from a client to play a stream, the stream service 224 needs to locate the Downstream Manager 210 associated with that client in order to give it indexing commands. The stream service 224 obtains a mapping between the client and its downstream manager from the Connection Service 230.

Detailed Description Text (177):
Once the client has established a virtual circuit through the media server 100, the client can send data to and receive data from services in the media server 100. However, the client still lacks its system software (unless it is stored in ROM) and the client network software. The client acquires any system software it needs using the Set-top Boot Protocol, which is implemented by the boot service.

Detailed Description Text (179):
1. The set-top box sends a boot request to the media server 100.

Detailed Description Text (180):
2. The upstream manager 220 and the connection service 230 cooperate to communicate addressing information to the boot service 216.

Detailed Description Text (187):
The name server 222 provides a global repository of string information that is accessible from any node. The name server 222 consists of the name server process and a programmatic interface which links into the processes that communicate with the name server. Processes obtain information about other services from the name server 222.

Detailed Description Text (190):
To accommodate many set-top boxes, many instances of the stream service 224 can run on the server 100. The stream service 224 can run in either single-instance mode or concurrent mode. In single-instance mode, there is only one instance of the service. In concurrent mode, many instances of the stream service run concurrently to accommodate many set-top boxes. Messages to the stream service 224 are received by an instance controller 229 and then routed to an available stream service instance. After the stream service instance has handled the message, the instance notifies the instance controller 229 that the instance is available to handle another message. Each stream service 224 instance can accommodate roughly 100 set-top boxes. By providing multiple instances of the stream service 224, the load on

the media server 100 can be more effectively balanced.

Detailed Description Text (192):
The authentication service 226 maintains an authentication database that keeps track of authorized media server 100 users, households, and set-top boxes 110. The authentication service 226 accesses this authentication database to verify authorization for both hardware and users. For example, when a set-top box 110 initially requests a connection to the media server 100, the authentication service 226 verifies that the box 110 box 110 and household where it is located are registered for connection to the media server 100.

Detailed Description Text (195):
Application services 240 receive messages from the upstream manager 220 and send messages to system services (such as the stream service 224) and other application services on behalf of applications running on set-top boxes 110. The application services 240 encompass many different types of application services including math services 242, home shopping services 244, movies on demand services 246, news on demand services 248 or other application specific services 249. These services are implemented with standard interfaces so the services are generally accessed in the same manner by a client or other server 100 services.

Detailed Description Text (201):
The real-time scheduler 299 controls access into the real-time section of the server 100. Real-time scheduler 299 receives access requests for real-time service and determines the loading impact the access will have on the system. The real-time scheduler 299 determines if the request can be granted given the current server load. If the request can be granted, the access is scheduled. The real-time scheduler 299 takes CPU, disk, and memory resources into account when analyzing a request.

Detailed Description Text (203):
The network management service 226 accepts routing commands during real-time stream playback. The network management service 226 determines if a stream needs to be re-routed given loading conditions and routing requests. If a stream needs to be re-routed, the network management service 226 commands the virtual network layer 212 to re-route the stream. Stream Server 224 is also notified of the new routing.

Detailed Description Text (205):
The media server 100 includes a performance monitoring function and capability for test and debug.

Detailed Description Text (207):
The media server 100 of the present invention uses the above described components to establish and control virtual connections between the media server 100 and an STB 110 as illustrated in FIGS. 6-12 and described below.

Detailed Description Text (208):
Referring now to FIG. 6, a block diagram illustrates the processing performed by the upstream manager 220 for establishing a virtual circuit. A client device 110 sends a request for service to upstream manager 220 on line 126. As described earlier, line 126 represents a data transfer through an arbitrary network. In a manner described below in connection with FIGS. 9 through 12, upstream manager 220 accesses connection service 230 to obtain a binding between the incoming request for service from line 126 and a downstream manager 210 for routing a response message or data stream on line 124 back to the client originating the request for service. Connection service 230 accesses a connection service table 320 to obtain information regarding the relationship between the upstream physical address, the client logical address, a corresponding downstream physical address, and a corresponding downstream logical address. When a virtual connection is initially established, connection service 230 allocates space for the new connection through

downstream manager 210. The connection information is stored in a routing table 310 and the connection service table 320. Once upstream manager 220 obtains a downstream manager 210 binding from connection service 230, the upstream manager 220 modifies the service request message received from client 110 to insert downstream manager binding information into the message. The service request message is then routed by upstream manager 220 to the requested media service 322 on line 324. Response messages or data stream activations are sent by the media service 322 directly to the downstream manager 210 identified by the binding information inserted into the service request message by upstream manager 220. In this manner, media service 322 is able to route information back to the client 110 through the previously bound downstream manager 210.

Detailed Description Text (209):
Referring now to FIG. 7, the structure of connection service table 320 is illustrated. Connection service table 320 comprises upstream manager physical address 410, client logical address 412, downstream manager physical address 414, downstream manager logical address 416, and downstream client physical address 418. Because the present invention is best used in an asymmetric network, the channel used by the client for transmitting service requests and information is not the same channel used by the client for receiving information from the server. The connection service table 320 is used to maintain information pertaining to the binding between the upstream channel and the downstream channel. Upstream manager physical address 410 is a port specific upstream physical address from which the upstream manager 220 receives data from a client 110. Client logical address 412 comprises an identifier of a client running on client device 110 which originates service requests for server 100 and consumes response messages and data streams received from media server 100. Downstream manager physical address 414 is a port-specific downstream physical address from which downstream manager 210 sends data on line 124 to a client device 110. Downstream manager logical address 416 uniquely identifies an instance of downstream manager lo 210 which is used for managing the channel identified by downstream manager physical address 414. Downstream client physical address 418 identifies the port-specific downstream address at which a client receives data from server 100. When a connection is initially established between a client 110 and server 100, the information in connection service table 320 is initialized to represent the binding of a complete set of links between a clients upstream and downstream channels. It will be apparent to those of ordinary skill in the art that additional connection information may be maintained in connection service table 320.

Detailed Description Text (210):
Referring now to FIG. 8, the routing table 310 is illustrated. Routing table 310 maintains information which associates a particular downstream physical address 422 with a corresponding downstream logical address 420. Because instances of downstream manager 210 may control a plurality of downstream manager physical addresses, routing table 310 is needed to maintain the association between instances of downstream manger 210 and the downstream physical addresses they control. Thus, routing table 310 comprises a downstream logical address 420 and corresponding downstream physical addresses 422 which are associated with the particular downstream logical address 420. Routing table 310 is built when the server 100 is initialized at system start-up.

Detailed Description Text (211):
Referring now to FIGS. 9 through 12, flow charts illustrate the processing logic performed by upstream manager 220 and connection service 230 of server 100 when a connection with a client 110 is first established and subsequently accessed in a request for service by client 110 through server 100.

Detailed Description Text (212):
Referring now to FIG. 9, the processing logic for establishing a connection from client 110 to server 100 is illustrated. In an initial message sent by client 110

to upstream manager 220, the client issues a request for initialization to media server 100 (processing block 612). In processing block 614, upstream manager 220 obtains the upstream physical address from a lower network layer of server 100 as the message from the client enters the media server 100. The upstream physical address identifies the client running on client device 110 that has originated the request to establish a connection with server 100. The upstream manager 220 is now aware of the client upstream physical address. The upstream manager 220 calls connection service 230 to request a connection for the client originating the connection request. Upstream manager 220 provides the upstream physical address to connection service 230 on line 326 (processing block 616). Connection service 230 accesses downstream manager 210 to request an instance of downstream manager 210 and a downstream physical address for the requesting client (processing block 618). The downstream manager 210 causes the allocation of a client logical address corresponding to the client. This client logical address is then returned to the client. Processing then continues at the bubble labeled A illustrated in FIG. 10.

Detailed Description Text (213):
Referring now to FIG. 10, processing for establishing a connection between a client and server 100 continues at the bubble labeled A. Once the connection service 230 has obtained a downstream manager logical address and a downstream manager physical address from downstream manager 210, the connection service table 320 and the routing table 310 is updated to record the association between the client logical address, the upstream physical address, the downstream manager logical address, and the downstream manager physical address. Having established the connection (processing block 710), processing for establishing a connection terminates through the exit bubble illustrated in FIG. 10.

Detailed Description Text (214):
Referring now to FIG. 11, processing logic by which a client on client device 110 accesses a service on server 100 is illustrated. The client sends a message via upstream manager 220 to request a service on server 100. The client provides the client logical address in the service request message initially received by upstream manager 220. The client also provides the logical address of the server that is the destination of the message. The message is broken down into packets and sent to the upstream manager 220 on the upstream channel 126. The upstream manager 220 simply routes these packets to the requested service using the logical destination address provided by the client. The message still carries the client logic address of the client that originated the message (processing block 810). Processing for accessing a service then continues through the bubble labeled B illustrated in FIG. 12.

Detailed Description Text (220):
Referring now to FIG. 16, the remote procedure call (RPC) mechanism of the present invention is illustrated. Because of the distributed nature of the present invention, client programs cannot be linked directly with the services that they use. Instead, the media server 100 of the present invention uses a remote procedure call (RPC) mechanism to communicate requests from clients to servers and to pass information back. To use the RPC mechanism, the client program is linked with a stub routine, which provides a function-call interface to the service. The stub routines shelter the client and server code from the complexities of machine dependence and logical network transport.

Detailed Description Text (223):
Step 2. The client stub routine 1312, 1) marshals data into a machine-independent form, and 2) uses the network protocol 1332 of the present invention to transmit the RPC information to the server stub routine 1318.

Detailed Description Text (224):
Step 3. On the server side, the RPC mechanism invokes the server stub routine 1318 on path 1315.

Detailed Description Text (225):
Step 4. The server stub routine 1318, 1) unmarshals the data, and 2) calls the actual server function 1316 on path 1317. Thus, the server function 1316 is invoked as if it has been called by a local routine in the client. The sequence of events is similar when the server returns data to the client.

Other Reference Publication (1):
IBM Almaden Research Center, The Shark Continuous-Media File Server, Roger L. Haskin.

CLAIMS:

1. A high bandwidth, scalable server for storing, retrieving, and transporting multimedia data to a client in a networked system, said server comprising:

an upstream manager receiving messages from said client and routing said messages to an appropriate service on said server, said upstream manager being coupled to a first network;

a downstream manager sending a stream of said multimedia data from said appropriate service on said server to said client, said downstream manager being coupled to a second network; and

a connection service for maintaining information to connect said client, said upstream manager, said downstream manager, and said appropriate service on said server.

2. The server in claim 1 wherein said connection service further creates a virtual connection between an upstream address and a downstream address for said client.

3. The server in claim 2 wherein said connection service also manages said virtual connection.

4. A computer-implemented method for retrieving and transporting multimedia data between a client and a server on a network, said computer-implemented method comprising the steps of:

receiving a client request for initialization in a message to an upstream manager in said server, said upstream manager being coupled to a first network;

obtaining an upstream physical address for said client as said client request enters said server;

allocating a downstream physical address and downstream logical address to said client corresponding to the upstream physical address obtained for said client, said downstream physical address being used by a downstream manager for sending a stream of said multimedia data from a service on said server to said client, said downstream manager being coupled to a second network; and

updating a connection service table with said upstream physical address, said downstream physical address, and said downstream logical address for said client.

5. The computer-implemented method in claim 4 wherein further comprising the steps of:

receiving a service request message from said client to said server via said upstream manager, said service request corresponding to said service on said server, said service request message including said client downstream logical address and a service destination logical address;

generating a response message from said server to said client, said response message including said client downstream logical address; and

sending said response message to said client via said downstream manager.

6. The computer-implemented method in claim 4 wherein said step of updating said connection service with said upstream and downstream addresses for said client includes the step of creating a virtual connection between said upstream and downstream addresses for said client.

7. The computer-implemented method in claim 6 wherein said step of creating said virtual connection between said upstream and downstream addresses for said client further includes the step of managing said virtual connection.

8. The computer-implemented method in claim 7 wherein said step of managing said virtual connection includes the steps of:

creating a routing table containing said client downstream logical address and a corresponding client downstream physical address;

accessing said connection service table; and

utilizing information in said routing table and said connection service table to route said client service request message from said client to said service in said server and to route said response message from said service in said server to said client via said downstream manager.

10. A high bandwidth, scalable server for storing, retrieving, and transporting multimedia data to a client in a networked system, said server comprising:

means for receiving a client request for initialization in a message to an upstream manager in said server, said upstream manager being coupled to a first network;

means for obtaining an upstream physical address for said client as said client request enters said server;

means for allocating a downstream physical address and downstream logical address for said client corresponding to the upstream physical address obtained for said client, said downstream physical address being used by a downstream manager for sending a stream of said multimedia data from a service on said server to said client, said downstream manager being coupled to a second network; and

means for updating a connection service table with said upstream physical address, said downstream physical address, and said downstream logical address for said client.

11. The server as claimed in claim 10 further including:

means for receiving a service request message from said client via said upstream manager, said service request corresponding to said service on said server, said service request message including said client downstream logical address and a service destination logical address;

means for generating a response message to said client, said response message including said client downstream logical address; and

means for sending said response message to said client via said downstream manager.

12. The server as claimed in claim 10 further including:

means for creating and managing a virtual connection between said upstream and downstream addresses for said client.

13. The server as claimed in claim 12 wherein said means for creating and managing said virtual connection further includes:

means for creating a routing table containing said client downstream logical address and a corresponding client downstream physical address;

means for accessing said connection service table; and

means for utilizing information in said routing table and said connection service table to route said client service request message from said client to said service in said server and to route said response message from said service in said server to said client via said downstream manager.

14. The server as claimed in claim 10 wherein said means for receiving a client request for initialization further includes a means for receiving a Remote Procedure Call (RPC).